Penny-board Plug-in Manual

Honours

Dannielle G. Smith

2101323

Table of Contents

Setting up the tool

```
#include "Penny-Board/PennyRender.h"
```

```
    pen::PennyRender screen_Renderer_;
    pen::PennyRender scene_Renderer_;
```

Make sure to initialise the renders in the h file of the game level your using.

```
screen_Renderer_.Init();
scene_Renderer_.Init();
```

Initialize the renders in the level init function, as shown above.

```
Scene::OnRender();

OnRender_firtPass();

screen_Renderer_.renderScreen(scene_Renderer_.getCurrentOutputSRVData().Get());
//test_Renderer2.MultiRenderPass(scene_Renderer_.getCurrentOutputSRVData().Get());
screen_Renderer_.renderToScreen_end(scene_Renderer_.getCurrentOutputRTVData(), scene_Renderer_.getCurrentOutputSRVData());
```

Have the screen used for post-processing effects drawn in the renderer, like shown in the above img.

```
scene_Renderer_.SetRenderTargets_(scene_Renderer_.getCurrentOutputTexture() , scene_Renderer_.getCurrentOutputRTVData());

scene_Renderer_.Begin();

//--
//scene stuff -- user can make scene here and not go within the tool to change anything
uint32_t Indexdata[]{0,1,2};
std::vector<pen::Vertex> movedvertices = vertices;
pen::InstanceData testdata;
testdata.TextureIndex = renderTexture.get()->GetViewIndex();

//draw triangle
for (auto& a : movedvertices) { a.Position += float3(2.5, 0, 0); }
scene_Renderer_.DrawVertices(movedvertices.data(), 3, Indexdata, 3, &testdata);

matrix World = glm::translate(float3(-5, 0, 0));
testdata.World = World * glm::translate(float3(5, -5 + Ypos, 0)) * glm::eulerAngleXYZ(glm::radians(Rotation.x), glm::radians
testdata.ColourScale = float4(1, 1, 1, 0.9);
testdata.SpecularColor = SpecularC;
testdata.SpecularPower = SpecularPower;
testdata.SpecularWeight = SpecularWeight;


pen::Vertex Quad[4] =
{
        {float3(-1,1,0),float3(0,0,0),float2(0,0),      float3(0,0,-1)  },
        {float3(-1,-1,0),float3(0,0,0),float2(0,1),     float3(0,0,-1)  },
        {float3(1,-1,0),float3(0,0,0),float2(1,1),      float3(0,0,-1)  },
        {float3(1,1,0),float3(0,0,0),float2(1,0),       float3(0,0,-1)  }
};

uint32_t quadindices[6] =
{
        0,1,2,2,3,0
};

scene_Renderer_.DrawVertices(Quad, 4, quadindices, 6, &testdata);

//--

scene_Renderer_.End();

scene_Renderer_.SetRenderTarget_ToBackBuffer();
```

Make sure to set the render target as shown above before the render begin, after the begin you can create the scene/ game environment then end the scene and set the render taget to the back buffer.

```
screen_Renderer_.PennyBoardIMGUI();
```

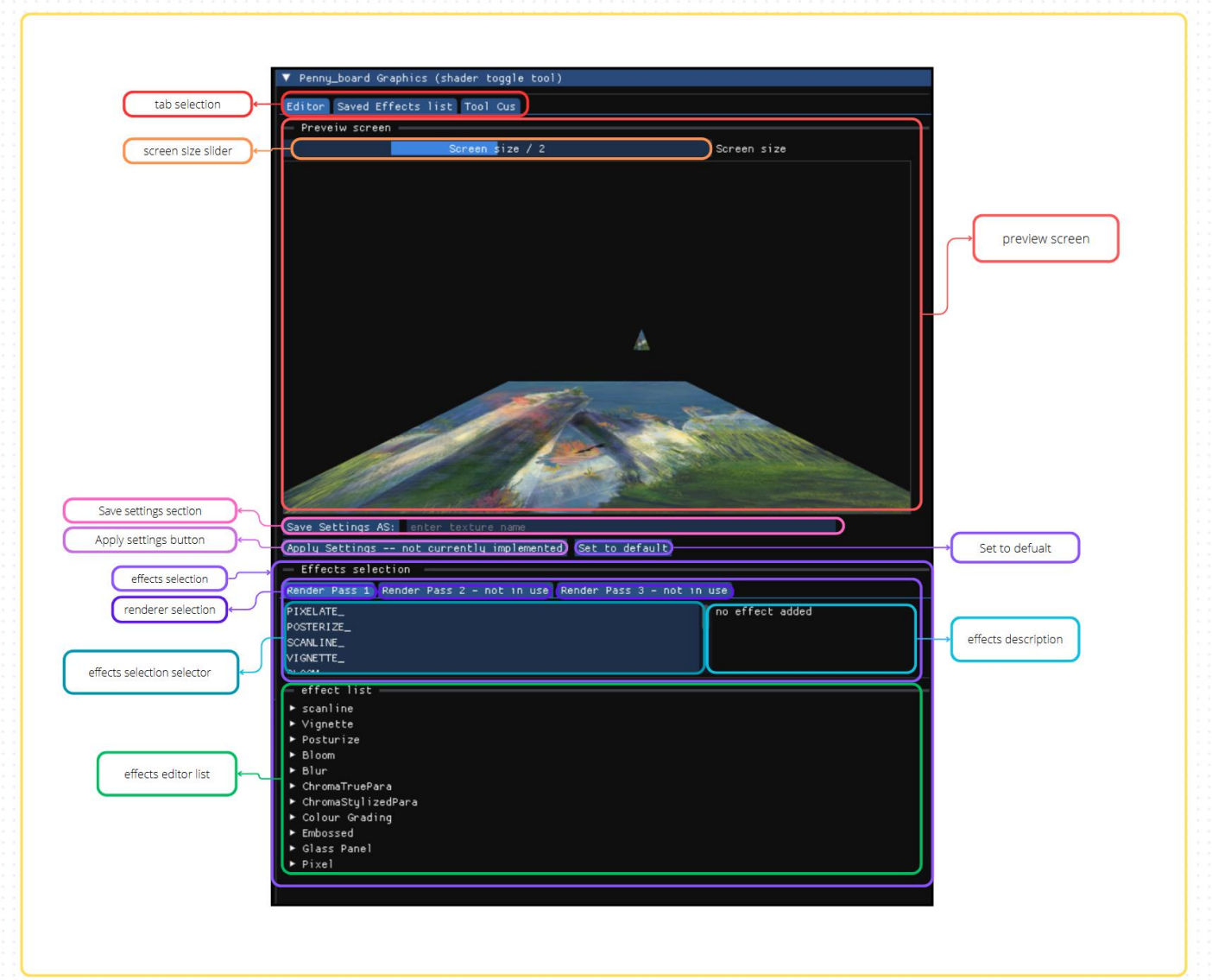
Have the tools GUI added to the IMGUI function in the levels code.

The tool should now work.

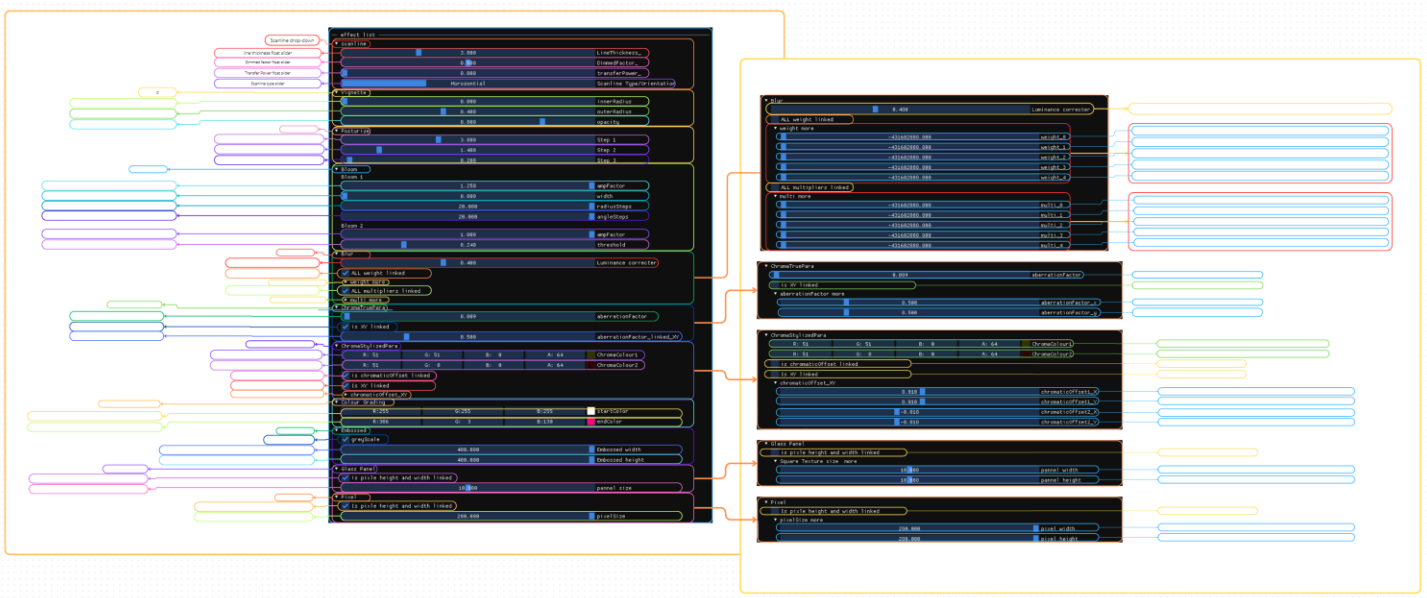
GUI

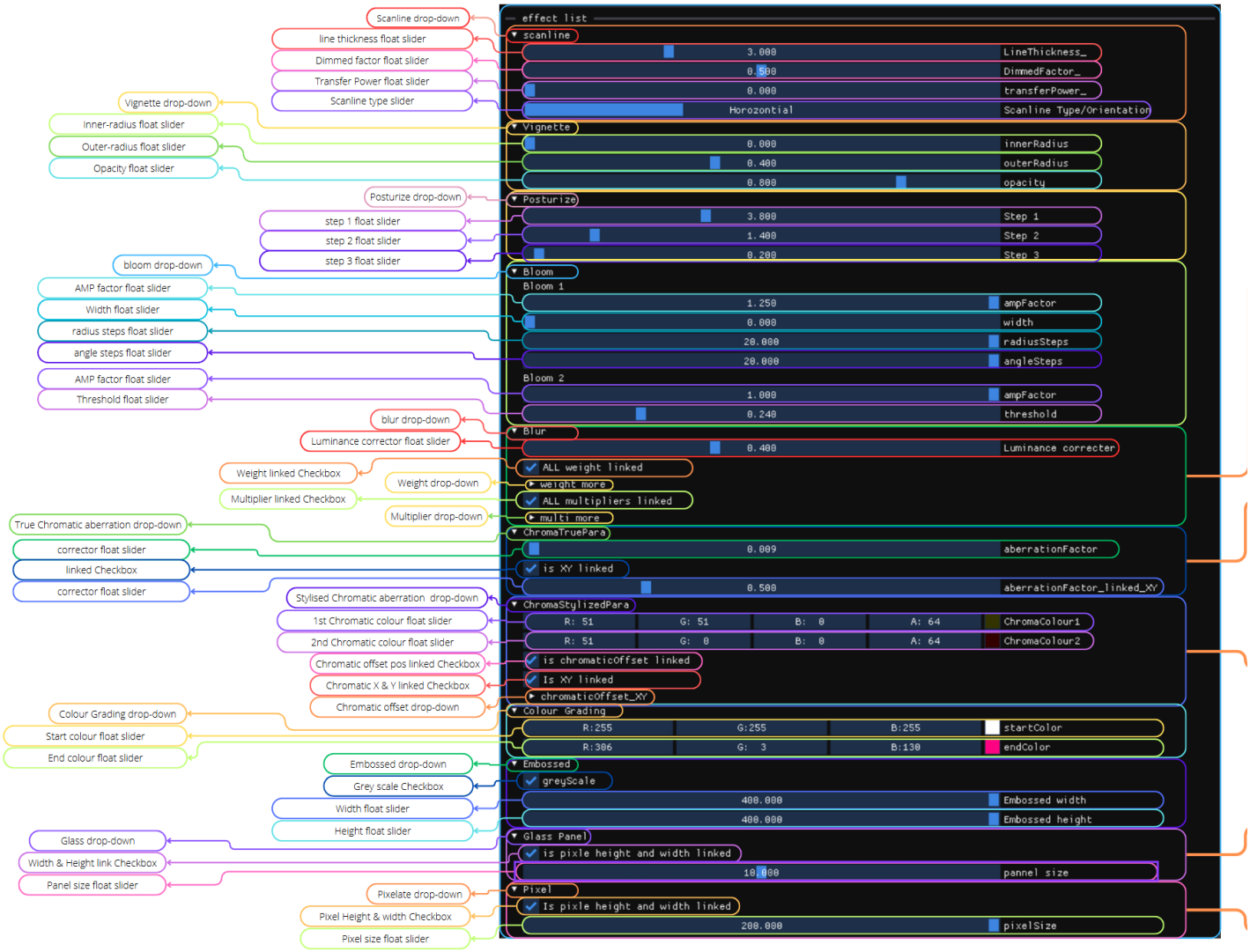
GUI layout

Editor tabs

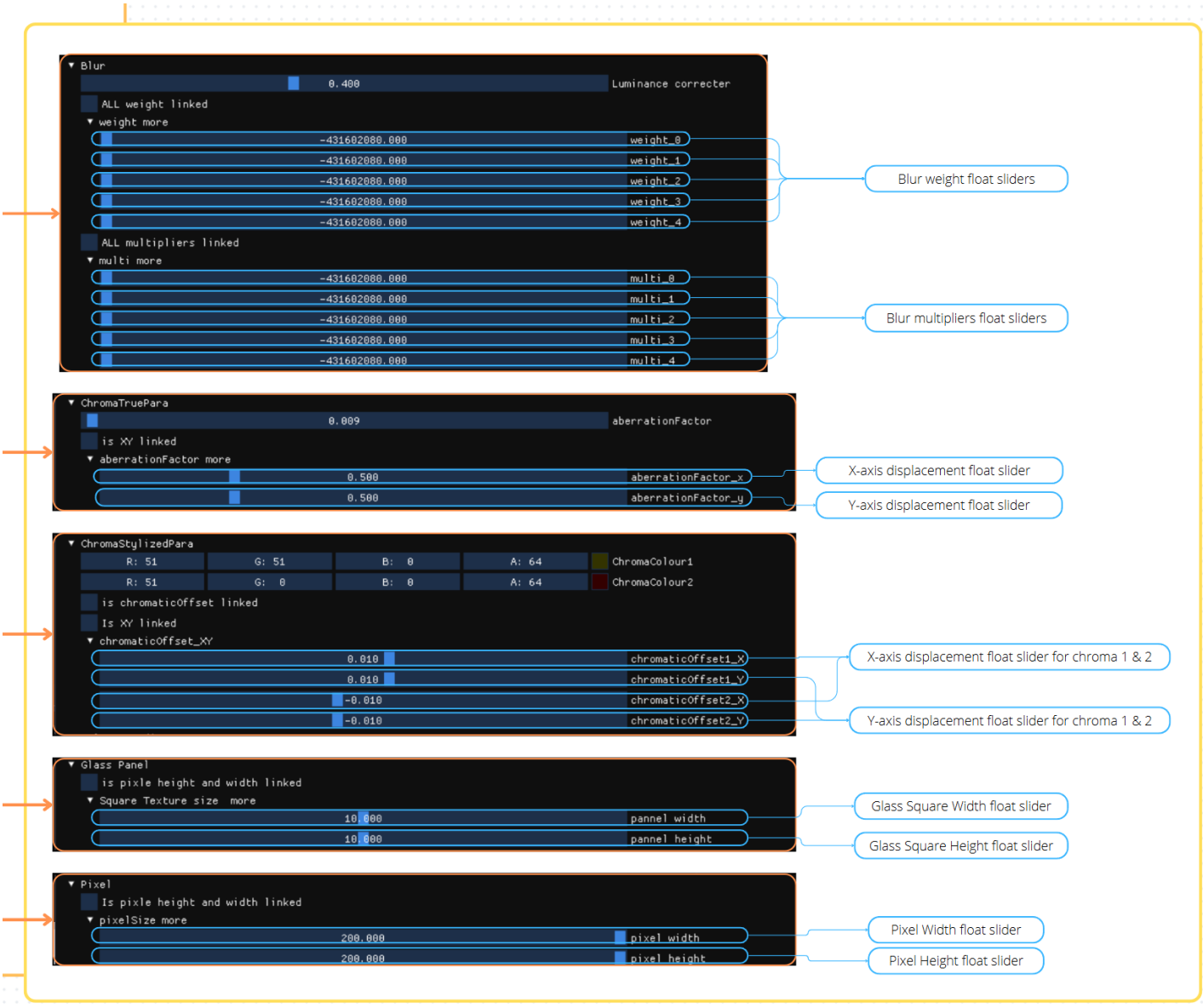


Selected area	Description
Preview screen	A small screen that shows the current settings.
Save settings As - button	Save current settings with the name given in the adjoining box.
Save settings - name space	Text box that contains the tag/name the user types in.
Apply settings - button	Apply current settings to the game screen, not just the preview screen.
Set to default - button	Set all settings to their default.
Effects in use list - section	The section where all the post-processing effects are toggled on or off.
- Offset sliders	Set the offset wanted on the chromatic aberration. There is one for the x-axis and one for the y.
- Sample No. slider	Slide the slider to choose the number of samples the down sampler will take.
-Colour picker	Pick the colour you want to use for the chromatic aberration effect.

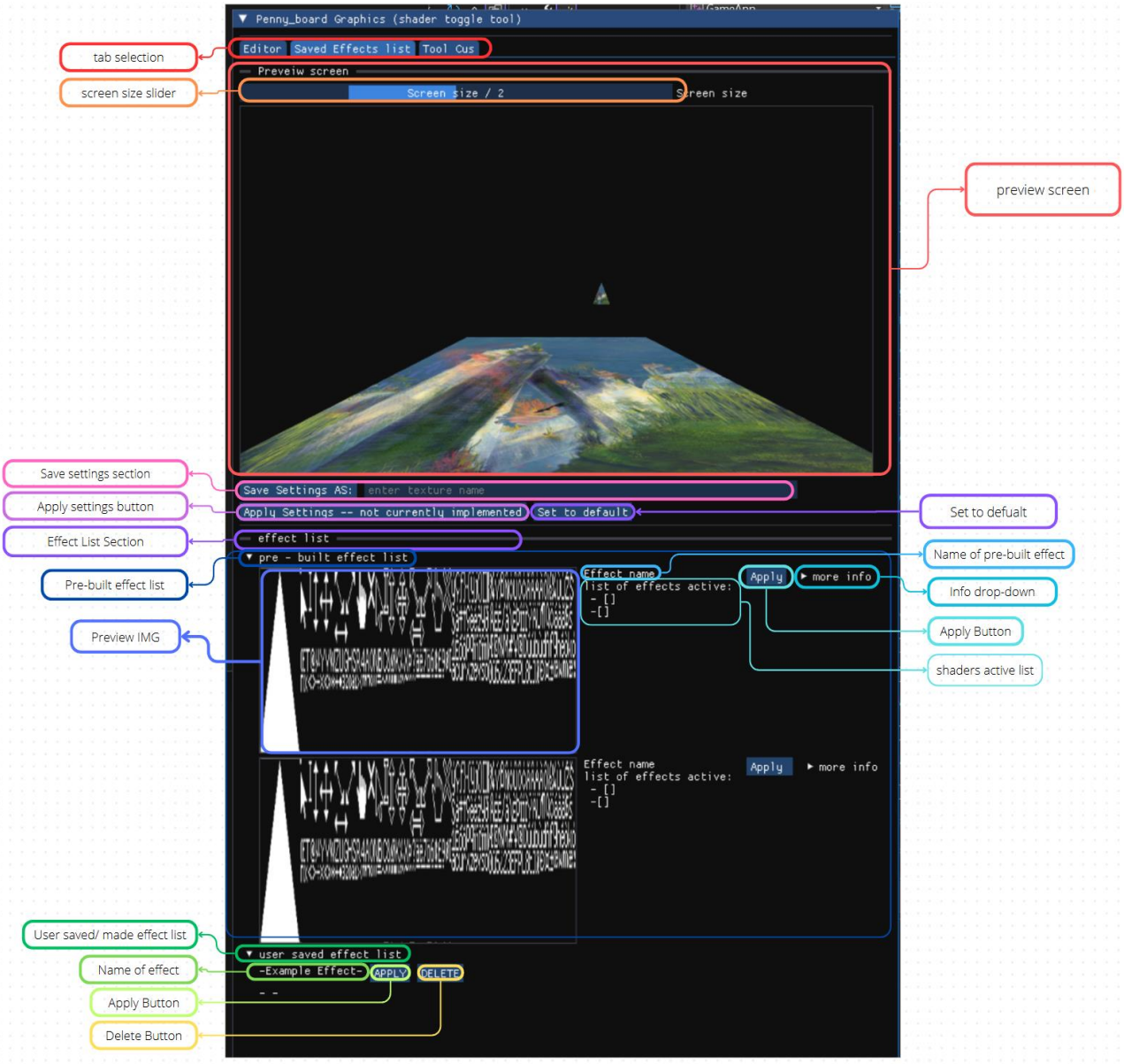




Selected area	Description

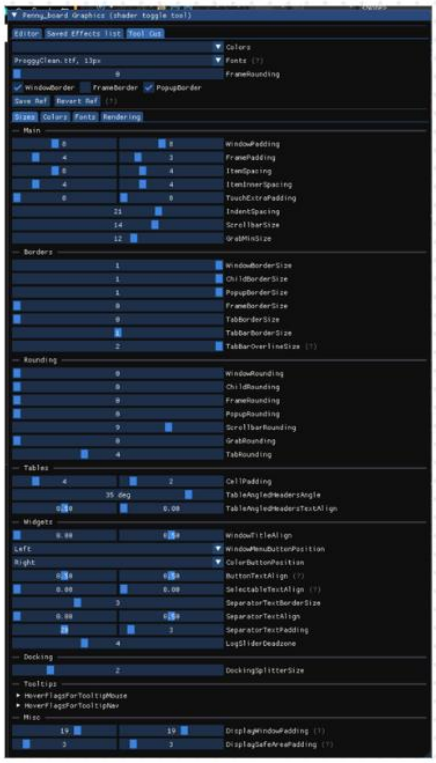

Blur
0.400
Luminance correcter
ALL weight linked
weight more
-431602080.000
weight_0
weight_1
weight_2
weight_3
weight_4
ALL multipliers linked
multi more
multi_0
multi_1
multi_2
multi_3
multi_4
Blur weight float sliders
Blur multipliers float sliders
ChromaTruePara
0.009
aberrationFactor
is XY linked
aberrationFactor more
0.500
aberrationFactor_x
aberrationFactor_y
X-axis displacement float slider
Y-axis displacement float slider
ChromaStylizedPara
R: 51
G: 51
B: 0
A: 64
ChromaColour1
R: 51
G: 0
B: 0
A: 64
ChromaColour2
is chromaticOffset linked
Is XY linked
chromaticOffset_XY
0.010
chromaticOffset1_X
chromaticOffset1_Y
-0.010
chromaticOffset2_X
chromaticOffset2_Y
X-axis displacement float slider for chroma 1 & 2
Y-axis displacement float slider for chroma 1 & 2
Glass Panel
is pixle height and width linked
Square Texture size more
10.000
pannel width
pannel height
Glass Square Width float slider
Glass Square Height float slider
Pixel
Is pixle height and width linked
pixelSize more
200.000
pixel width
pixel height
Pixel Width float slider
Pixel Height float slider

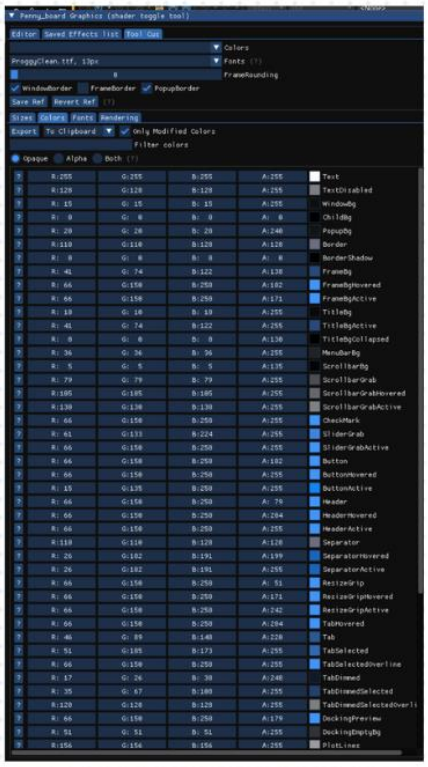
Saved lists tab

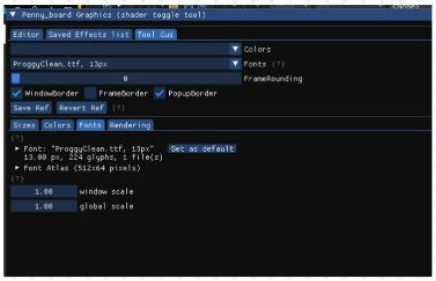


Selected area	Description
Preview screen	A small screen that shows the current settings.
Save settings As - button	Save current settings with the name given in the adjoining box.
Save settings - name space	Text box that contains the tag/name the user types in.
Apply settings - button	Apply current settings to the game screen, not just the preview screen.
Set to default - button	Set all settings to their default.
Effects list - section	-

Tool customizer tab







How the GUI is set-up

//--

```
void PennyRender::PennyBoardIMGUI_PreveiwScreen() {
        //-- barriers to tye/sync render target
        TextureBarrier renderTargetBarrier{};
        renderTargetBarrier.SyncBefore = SKTBD_SYNC_RENDER_TARGET;
        renderTargetBarrier.SyncAfter = SKTBD_SYNC_PIXEL_SHADING;
        renderTargetBarrier.AccessBefore = SKTBD_ACCESS_RENDER_TARGET;
        renderTargetBarrier.AccessAfter = SKTBD_ACCESS_COMMON;
        renderTargetBarrier.LayoutBefore = SKTBD_LAYOUT_RENDER_TARGET;
        renderTargetBarrier.LayoutAfter = SKTBD_LAYOUT_COMMON;
        renderTargetBarrier.Resource = OutputTexture.Get().get();
        renderTargetBarrier.SubresourceRange = TextureSubresourceRange();//

        BarrierGroup grouptoRTV(&renderTargetBarrier);
        RenderCommand::Barrier(&grouptoRTV, 1);
        //--

        ImGui::SeparatorText("Preveiw screen");

        ImGui::SliderInt("Screen size", &previewscreenSize, 1, 4, "Screen size / %d%");
        ImGui::SetItemTooltip("Preveiw screen size, Fullsize/ n");

        //preveiw screen
        ImGuiIO& io = ImGui::GetIO();
        ImTextureID my_tex_id = OutputSRV.Get()->GetImTextureID();//turning srv to a usable ImTextureID - for preview screen
        //move to next frame
        renderToScreen_end(OutputRTV, OutputSRV);

        float my_tex_w = (float)1366 / previewscreenSize;
        float my_tex_h = (float)768 / previewscreenSize;
        {
                static bool use_text_color_for_tint = false;

                ImVec2 uv_min = ImVec2(0.0f, 0.0f);                 // Top-left
                ImVec2 uv_max = ImVec2(1.0f, 1.0f);                 // Lower-right
                ImVec4 tint_col = use_text_color_for_tint ? ImGui::GetStyleColorVec4(ImGuiCol_Text) : ImVec4(1.0f, 1.0f, 1.0f, 1.0f);   // No tint
                ImVec4 border_col = ImGui::GetStyleColorVec4(ImGuiCol_Border);
                ImGui::Image(my_tex_id, ImVec2(my_tex_w, my_tex_h), uv_min, uv_max, tint_col, border_col);
                ImGui::SetItemTooltip("Preveiw screen;\nshow the effected in development before adding to the proper screen/view");
        }
```

```
        //button settings/saves
        static int clicked = 0;
        if (ImGui::Button("Save Settings AS:")) {
                clicked++;
        }
        ImGui::SetItemTooltip("Click button to Save your current effect settings.");

        ImGui::SameLine();
        static char str1[32] = "";
        ImGui::InputTextWithHint(" ", "enter texture name", str1, IM_ARRAYSIZE(str1));
        ImGui::SetItemTooltip("Type in the name you want to save your current effect as. \n for example 'DarkRoom_1' \n to find these effects layter on go to the 'saved effects' tab on the top of the tool screen");

        if (clicked & 1)
        {
                ImGui::SameLine();
                //ImGui::Text("settings saved");

                if (savedEffect_List.size() > 4) /*max4 for now*/{
                        ImGui::Text("delete an effect for more space"); std::cout << "delete an effect for more space\n"; /*have a pop-up layter*/

                }
                else {
                        ImGui::Text("settings saved");
                        SavedEffect currentEffect = ConvertToSavedEffect(str1, CURRENT_STATE_,m_ScreenBuffer,m_ScanlineBuffer,m_VignettingBuffer,
                                                                                                                m_PosterizeBuffer,m_BloomBuffer,m_Bloom2Buffer,m_BlurGauBuffer,m_ChromaTrueBuffer,
                                                                                                                m_ChromaStylizedBuffer,m_ColourGradBuffer,m_EmbossBuffer,m_GlassBuffer,m_PixelBuffer); // I know this is ridiculous, ill find a better way after submission

                        save_SavedEffect(currentEffect);

                        clicked = 0;
                }
        }

        //ImGui::SameLine();
        if (ImGui::Button("Apply Settings -- not currently implemented")) {/*effectValuesCurrentlyInUse = effectValuesPreveiw; */ }
        ImGui::SetItemTooltip("Apply current settings to the actual screen. \n This doesnt save the current settings To do that click the 'Save Settings AS:' button with your effects chosen name.");

        ImGui::SameLine();
        if (ImGui::Button("Set to default")) { ResetAll();}
        ImGui::SetItemTooltip("This sets the current values/settings back to when you last clicked 'Apply settings'");
        //
}
```

//--

```
void PennyRender::PennyBoardIMGUI_PassApply(int PassNo){

	const char* items[] = { "PIXELATE_", "POSTERIZE_", "SCANLINE_", "VIGNETTE_",
		"BLOOM_", "CHROMABER_TRUE_", "CHROMABER_STYLE_", "COLOURGRAD_", "GLASS_", "EMBOS_", "BLUR_GAU_","n/a"};

	const char* items_desc[] = {
		"PIXELATE_ \n Turns given veiw into a pixelated version, like looking at a lower resolution",
		"POSTERIZE_\n Reduces the number of colors in an image, creating a stylized, high-contrast look with distinct color bands or gradients.",
		"SCANLINE_\n Simulates the appearance of lines running across the screen, for a retro or vintage CRT monitor look by adding subtle dark bands to an image.",
		"VIGNETTE_\n Darkens the edges of an image while keeping the center bright, drawing attention to the focal point and creating a more dramatic or cinematic atmosphere.",
		"BLOOM_ \n Creates a glowing haze around the bright areas of an image, enhancing light sources.",
		"CHROMABER_TRUE_\n Simulates color fringing by slightly offsetting the red, green, and blue channels along the edges of objects, similar to what 3D movies look like without the glasses.",
		"CHROMABER_STYLE_ \n Simalar to true Chromatic aberation, this effect mimics the fringing whilst having the colours used customiseable giving a unique look. ",
		"COLOURGRAD_\n Adjusts the colours and tones of an image to create a specific mood or aesthetic, enhancing visual storytelling through subtle shifts in hue, saturation, and contrast.",
		"GLASS_ \n Simulates the appearance of looking through panelled textured glass",
		"EMBOS_ \n Creates the illusion of raised or recessed surfaces on an image, adding depth and texture by highlighting edges with a contrast of light and shadow.",
		"BLUR_GAU_ \n Smooths the image by averaging pixel values based on a Gaussian distribution, creating a soft, evenly blurred effect.",
		"no effect added"};
```

```
		static int item_current = 11;
		ImGui::ListBox(items_desc[item_current], &item_current, items, IM_ARRAYSIZE(items), 4);
		//ImGui::ListBox("listbox", &PassNo, items, IM_ARRAYSIZE(items), 4);

		switch (item_current)
		{
		case 0:
			//CURRENT_STATE_ = PIXELATE_;
			SetCurrentState(PIXELATE_);
			break;
		case 1:
			//CURRENT_STATE_ = POSTERIZE_;
			SetCurrentState(POSTERIZE_);
			break;
		case 2:
			//CURRENT_STATE_ = SCANLINE_;
			SetCurrentState(SCANLINE_);
			break;
		case 3:
			//CURRENT_STATE_ = VIGNETTE_;
			SetCurrentState(VIGNETTE_);
			break;
		case 4:
			//RSPD.SetPixelShader(L"bloom_ps");
			//CURRENT_STATE_ = BLOOM_;
			SetCurrentState(BLOOM_);
			break;
		case 5:
			//CURRENT_STATE_ = CHROMABER_TRUE_;
			SetCurrentState(CHROMABER_TRUE_);
			break;
		case 6:
			//CURRENT_STATE_ = CHROMABER_STYLE_;
			SetCurrentState(CHROMABER_STYLE_);
			break;
		case 7:
			//CURRENT_STATE_ = COLOURGRAD_;
			SetCurrentState(COLOURGRAD_);
			break;
		case 8:
			//CURRENT_STATE_ = GLASS_;
			SetCurrentState(GLASS_);
			break;
		case 9:
			//CURRENT_STATE_ = EMBOS_;
			SetCurrentState(EMBOS_);
			break;
		case 10:
			//CURRENT_STATE_ = BLUR_GAU_;
			SetCurrentState(BLUR_GAU_);
			break;

		default:
			//CURRENT_STATE_ = 11;
			SetCurrentState(POSTPROSSESSING_);
			break;
		}

		m_BlurGauBuffer;
	}
```

///---

```
void PennyRender::PennyBoardIMGUI_editorTab()
{
        PennyBoardIMGUI_PreveiwScreen();

        ImGui::SeparatorText("Effects selection ");
        ImGui::SetItemTooltip("Where you can select effects in any given order using 'render passes' to create you unique effect ");

        if (ImGui::BeginTabBar("##tabs", ImGuiTabBarFlags_None))
        {
                if (ImGui::BeginTabItem("Render Pass 1")) {

                        PennyBoardIMGUI_PassApply(0);

                        ImGui::EndTabItem();

                }
                if (ImGui::BeginTabItem("Render Pass 2 - not in use")) {
                        PennyBoardIMGUI_PassApply(1);

                        ImGui::EndTabItem();

                }
                if (ImGui::BeginTabItem("Render Pass 3 - not in use")) {
                        PennyBoardIMGUI_PassApply(2);

                        ImGui::EndTabItem();

                }

                ImGui::EndTabBar();
        }
```

```
{
        ImGui::Separator();

        ImGui::SeparatorText("effect list");
        ImGui::SetItemTooltip("Lists all effects and allows user to edit said effects");

        if (ImGui::TreeNode("scanline"))
        {
                //ImGui::Text("scanline");

                ImGui::SliderFloat("LineThickness_", (float*)&m_ScanlineBuffer.LineThickness_, 0, 10);
                ImGui::SetItemTooltip("The thickness of the bands that have a tint/saturation change.");
                ImGui::SliderFloat("DimmedFactor_", (float*)&m_ScanlineBuffer.DimmedFactor_, 0, 1);
                ImGui::SetItemTooltip("How 'dimmed' the darker screen bands are");
                ImGui::SliderFloat("transferPower_", (float*)&m_ScanlineBuffer.transferPower_, 0, 1);
                ImGui::SetItemTooltip("The 'brightness' of the lighter screen bands");

                enum Element { scanline_H, scanline_C, scanline_V, scanline_COUNT };
                static int scanline_Type = scanline_H;
                const char* scanline_Type_names[scanline_COUNT] = { "Horozontial", "Cross/Matrix", "Vertical" };
                const char* scanline_Type_name = (scanline_Type >= 0 && scanline_Type < scanline_COUNT) ? scanline_Type_names[scanline_Type] : "Unknown";
                ImGui::SliderInt("Scanline Type/Orientation", &scanline_Type, 0, scanline_COUNT - 1, scanline_Type_name); // Use ImGuiSliderFlags_NoInput
                ImGui::SetItemTooltip("The Type/Orientation of the screen bands/lines (Horozontial,Vertical,Cross/Matrix)");

                switch (scanline_Type)
                {
                case scanline_H:
                        m_ScanlineBuffer.vertical_ = 0;
                        //vertical_ = 0;
                        break;
                case scanline_C:
                        m_ScanlineBuffer.vertical_ = 0.5;
                        //vertical_ = 0.5;
                        break;
                case scanline_V:
                        m_ScanlineBuffer.vertical_ = 1;
                        //vertical_ = 1;
                        break;
                default:
                        scanline_Type = scanline_H;
                        break;
                }

                ImGui::TreePop();
        }
        ImGui::SetItemTooltip("Simulates the appearance of lines running across the screen, for a retro or vintage CRT monitor look by adding subtle dark bands to an image.");

        if (ImGui::TreeNode("Vignette")) {

                ImGui::SliderFloat("innerRadius", (float*)&m_VignettingBuffer.innerRadius, 0, 1);
                ImGui::SetItemTooltip("where the gradient of the vingette ends");
                ImGui::SliderFloat("outerRadius", (float*)&m_VignettingBuffer.outerRadius, 0, 1);
                ImGui::SetItemTooltip("where the gradient of the vingette starts");
                ImGui::SliderFloat("opacity", (float*)&m_VignettingBuffer.opacity, 0, 1);
                ImGui::SetItemTooltip("The opacity/transparency of the vignette boarder");
                ImGui::TreePop();
        }
        ImGui::SetItemTooltip("Darkens the edges of an image while keeping the center bright, drawing attention to the focal point and creating a more dramatic or cinematic atmosphere.");
```

```
if (ImGui::TreeNode("Posturize")) {

        ImGui::SliderFloat("Step 1", (float*)&m_PosterizeBuffer.step_areas.x, 0, 10);
        ImGui::SetItemTooltip("The first 'tonal level'/ band .");

        ImGui::SliderFloat("Step 2", (float*)&m_PosterizeBuffer.step_areas.y, 0, 10);
        ImGui::SetItemTooltip("The second 'tonal level'/ band .");

        ImGui::SliderFloat("Step 3", (float*)&m_PosterizeBuffer.step_areas.z, 0, 10);
        ImGui::SetItemTooltip("The third 'tonal level'/ band .");
        //number of tonal levels
        ImGui::TreePop();
}
ImGui::SetItemTooltip("Reduces the number of colors in an image, creating a stylized, high-contrast look with distinct color bands or gradients.");
//might need to change des

if (ImGui::TreeNode("Bloom")) {

        ImGui::Text("Bloom 1");

        ImGui::SliderFloat("ampFactor", (float*)&m_BloomBuffer.ampFactor, 0, 1);
        ImGui::SetItemTooltip("How bright the brightest areas of the screen are");
        ImGui::SliderFloat("width", (float*)&m_BloomBuffer.width, 0, 10);
        ImGui::SetItemTooltip("How wide the blur radius will be");
        ImGui::SliderFloat("radiusSteps", (float*)&m_BloomBuffer.radiusSteps, 0, 10);
        ImGui::SetItemTooltip("---");
        ImGui::SliderFloat("angleSteps", (float*)&m_BloomBuffer.angleSteps, 0, 10);
        ImGui::SetItemTooltip("---");

        ImGui::Text("Bloom 2");//this is more a threshold thing at this point

        ImGui::SliderFloat("ampFactor ", (float*)&m_Bloom2Buffer.ampFactor, 0, 1);
        ImGui::SetItemTooltip("How bright the brightest areas of the screen are");
        ImGui::SliderFloat("threshold", (float*)&m_Bloom2Buffer.threshold, 0, 1);
        ImGui::SetItemTooltip("This decides what the 'brightesr' areas of the view are.");//should i describe the whole colour number thing or would that be a bit much?

        ImGui::TreePop();
}
ImGui::SetItemTooltip("Creates a glowing haze around bright areas of an image, enhancing light sources.");
```

```
if (ImGui::TreeNode("Blur")) {

	ImGui::SliderFloat("Luminance correcter", (float*)&m_BlurGauBuffer.Luminance, 0, 1);
	ImGui::SetItemTooltip("This correct/ changes the brightness of the produced effect.");

	ImGui::Checkbox("ALL weight linked", &link_weight);
	ImGui::SetItemTooltip("If yes the blur will be even throughout the given veiw");
	if (link_weight == true) {

		ImGui::SliderFloat("weight_all", (float*)&weight_all, 0, 0.4);
		ImGui::SetItemTooltip("--");
		for (int i = 0; i < 5; i++) {
			m_BlurGauBuffer.weight[i] = weight_all;
		}
	}
	else {
		if (ImGui::TreeNode("weight more")) {

			ImGui::SliderFloat("weight_0", (float*)&m_BlurGauBuffer.weight[0], 0, 0.4);
			ImGui::SetItemTooltip("Center of the point being blurred/ the orign");
			ImGui::SliderFloat("weight_1", (float*)&m_BlurGauBuffer.weight[1], 0, 0.4);
			ImGui::SetItemTooltip("Middle point of blur, close to the orign of the blur");
			ImGui::SliderFloat("weight_2", (float*)&m_BlurGauBuffer.weight[2], 0, 0.4);
			ImGui::SetItemTooltip("Middle point of blur");
			ImGui::SliderFloat("weight_3", (float*)&m_BlurGauBuffer.weight[3], 0, 0.4);
			ImGui::SetItemTooltip("Middle point of blur, close to the edge of the blur");
			ImGui::SliderFloat("weight_4", (float*)&m_BlurGauBuffer.weight[4], 0, 0.4);
			ImGui::SetItemTooltip("Edge of the point being blurred");

			ImGui::TreePop();
		}
	}
```

```
        ImGui::Checkbox("ALL multipliers linked", &link_Multiplier);// maybe i shouldn't include this
        ImGui::SetItemTooltip("If yes the blur will not be very visable");
        if (link_Multiplier == true) {

                ImGui::SliderFloat("multi_all", (float*)&multi_all, 0, 10);
                ImGui::SetItemTooltip("The value used to multiply all blur points");
                for (int i = 0; i < 5; i++) {
                        m_BlurGauBuffer.texscrMultiplier[i] = multi_all;
                }
        }
        else {
                if (ImGui::TreeNode("multi more")) {

                        ImGui::SliderFloat("multi_0", (float*)&m_BlurGauBuffer.texscrMultiplier[0], 0, 1);
                        ImGui::SetItemTooltip("The value used to multiply the center of the point being blurred/ the orign");
                        ImGui::SliderFloat("multi_1", (float*)&m_BlurGauBuffer.texscrMultiplier[1], 0, 1);
                        ImGui::SetItemTooltip("The value used to multiply the middle point of blur, close to the orign of the blur");
                        ImGui::SliderFloat("multi_2", (float*)&m_BlurGauBuffer.texscrMultiplier[2], 0, 1);
                        ImGui::SetItemTooltip("The value used to multiply the middle point of blur");
                        ImGui::SliderFloat("multi_3", (float*)&m_BlurGauBuffer.texscrMultiplier[3], 0, 1);
                        ImGui::SetItemTooltip("The value used to multiply the middle point of blur, close to the edge of the blur");
                        ImGui::SliderFloat("multi_4", (float*)&m_BlurGauBuffer.texscrMultiplier[4], 0, 1);
                        ImGui::SetItemTooltip("The value used to multiply the edge of the point being blurred");

                        ImGui::TreePop();
                }
        }

        ImGui::TreePop();
}
ImGui::SetItemTooltip("Smooths the image by averaging pixel values based on a Gaussian distribution, creating a soft, evenly blurred effect.");
```

```
if (ImGui::TreeNode("ChromaTruePara")) {

        ImGui::SliderFloat("aberrationFactor", (float*)&m_ChromaTrueBuffer.aberrationFactor, 0, 1);
        ImGui::SetItemTooltip("The chromatic aberration layers will be linked and the values will be mirrored for the r,y and b values.");

        ImGui::Checkbox("is XY linked", &link_XY);
        ImGui::SetItemTooltip("Whether the chromatic aberration x-axis and y-axis are linked / 'mirrored' with each other.");//

        if (link_XY == true) {

                ImGui::SliderFloat("aberrationFactor_linked_XY", (float*)&aberrationFactor_XY, 0, 2);
                ImGui::SetItemTooltip("The diffrence betewn the r,g, and b values in terms of the x and y axis ");

                m_ChromaTrueBuffer.aberrationFactor_y = aberrationFactor_XY;
                m_ChromaTrueBuffer.aberrationFactor_x = aberrationFactor_XY;
        }
        else {
                if (ImGui::TreeNode("aberrationFactor more")) {

                        ImGui::SliderFloat("aberrationFactor_x", (float*)&m_ChromaTrueBuffer.aberrationFactor_x, 0, 2);
                        ImGui::SetItemTooltip("The diffrence betewn the r,g, and b values in terms of the x axis ");
                        ImGui::SliderFloat("aberrationFactor_y", (float*)&m_ChromaTrueBuffer.aberrationFactor_y, 0, 2);
                        ImGui::SetItemTooltip("The diffrence betewn the r,g, and b values in terms of they axis ");

                        ImGui::TreePop();
                }
        }
        ImGui::TreePop();
}
ImGui::SetItemTooltip("Simulates color fringing by slightly offsetting the red, green, and blue channels along the edges of objects, similar to what 3D movies look like");
```

```
if (ImGui::TreeNode("ChromaStylizedPara")) {

        chromaticWeights1[0] = { m_ChromaStylizedBuffer.chromaticWeights1.x };
        chromaticWeights1[1] = { m_ChromaStylizedBuffer.chromaticWeights1.y, };
        chromaticWeights1[3] = { m_ChromaStylizedBuffer.chromaticWeights1.a };

        chromaticWeights2[0] = { m_ChromaStylizedBuffer.chromaticWeights2.x };
        chromaticWeights2[1] = { m_ChromaStylizedBuffer.chromaticWeights2.y, };
        chromaticWeights2[3] = { m_ChromaStylizedBuffer.chromaticWeights2.a };

        ImGui::ColorEdit4("ChromaColour1 ", chromaticWeights1);
        ImGui::SetItemTooltip("The colour used for the first chromatic aberration layer");

        ImGui::ColorEdit4("ChromaColour2 ", chromaticWeights2);
        ImGui::SetItemTooltip("The colour used for the second chromatic aberration layer");

        m_ChromaStylizedBuffer.chromaticWeights1 = float4(chromaticWeights1[0], chromaticWeights1[1], 0.f, chromaticWeights1[3]);
        m_ChromaStylizedBuffer.chromaticWeights2 = float4(chromaticWeights2[0], chromaticWeights2[1], 0.f, chromaticWeights2[3]);

        ImGui::Checkbox("is chromaticOffset linked", &offsetlinked);
        ImGui::SetItemTooltip("The chromatic aberration layers will be linked and the values will be mirrored for both layers");
        ImGui::Checkbox("Is XY linked", &XYlinked);
        ImGui::SetItemTooltip("Whether the chromatic aberration x-axis and y-axis ar equal/mirrored with one another.");

        chromaticOffset_XY.x = m_ChromaStylizedBuffer.chromaticOffset1.x;
        chromaticOffset_XY.y = m_ChromaStylizedBuffer.chromaticOffset1.y;

        if (offsetlinked == true) {

                if (ImGui::TreeNode("chromaticOffset_XY")) {

                        if (XYlinked == true) {
                                ImGui::SliderFloat("chromaticOffset_XY", (float*)&chromaticOffset_XY.x, -0.1, 0.1);

                                chromaticOffset_XY.y = chromaticOffset_XY.x;
                                m_ChromaStylizedBuffer.chromaticOffset1.x = chromaticOffset_XY.x;
                                m_ChromaStylizedBuffer.chromaticOffset1.y = chromaticOffset_XY.y;

                                m_ChromaStylizedBuffer.chromaticOffset2.x = -chromaticOffset_XY.x;
                                m_ChromaStylizedBuffer.chromaticOffset2.y = -chromaticOffset_XY.y;
                        }
                        else {
                                ImGui::SliderFloat("chromaticOffset_X", (float*)&chromaticOffset_XY.x, -0.1, 0.1);
                                ImGui::SliderFloat("chromaticOffset_Y", (float*)&chromaticOffset_XY.y, -0.1, 0.1);

                                m_ChromaStylizedBuffer.chromaticOffset1.x = chromaticOffset_XY.x;
                                m_ChromaStylizedBuffer.chromaticOffset1.y = chromaticOffset_XY.y;

                                m_ChromaStylizedBuffer.chromaticOffset2.x = chromaticOffset_XY.x;
                                m_ChromaStylizedBuffer.chromaticOffset2.y = chromaticOffset_XY.y;
                        }

                        ImGui::TreePop();
                }
        }
```

```
        else {
                if (ImGui::TreeNode("chromaticOffset_XY")) {

                        if (XYlinked == true) {
                                ImGui::SliderFloat("chromaticOffset_1_XY", (float*)&m_ChromaStylizedBuffer.chromaticOffset1.x, -0.1, 0.1);
                                ImGui::SliderFloat("chromaticOffset_2_XY", (float*)&m_ChromaStylizedBuffer.chromaticOffset2.x, -0.1, 0.1);

                                m_ChromaStylizedBuffer.chromaticOffset1.y = m_ChromaStylizedBuffer.chromaticOffset1.x;
                                m_ChromaStylizedBuffer.chromaticOffset2.y = m_ChromaStylizedBuffer.chromaticOffset2.x;
                        }
                        else {
                                ImGui::SliderFloat("chromaticOffset1_X", (float*)&m_ChromaStylizedBuffer.chromaticOffset1.x, -0.1, 0.1);
                                ImGui::SliderFloat("chromaticOffset1_Y", (float*)&m_ChromaStylizedBuffer.chromaticOffset1.y, -0.1, 0.1);
                                ImGui::SliderFloat("chromaticOffset2_X", (float*)&m_ChromaStylizedBuffer.chromaticOffset2.x, -0.1, 0.1);
                                ImGui::SliderFloat("chromaticOffset2_Y", (float*)&m_ChromaStylizedBuffer.chromaticOffset2.y, -0.1, 0.1);
                        }
                        ImGui::TreePop();
                }
        }

        ImGui::TreePop();
}
ImGui::SetItemTooltip("Simalar to true Chromatic aberation, this effect mimics the fringing whilst having the colours used customiseable giving a unique look.");
```

```
if (ImGui::TreeNode("Colour Grading")) {

        startColor_CG[0] = { m_ColourGradBuffer.startColor.x };
        startColor_CG[1] = { m_ColourGradBuffer.startColor.y };
        startColor_CG[2] = { m_ColourGradBuffer.startColor.z };

        endColor_CG[0] = { m_ColourGradBuffer.endColor.x };
        endColor_CG[1] = { m_ColourGradBuffer.endColor.y };
        endColor_CG[2] = { m_ColourGradBuffer.endColor.z };

        ImGui::ColorEdit3("startColor ", startColor_CG);
        ImGui::SetItemTooltip("Start colour sets the baseline or beginning of a colour effect");
        //It's the colour at the "beginning" of the tonal range, often used to tint the shadows or define the base tone of the image.
        ImGui::ColorEdit3("endColor ", endColor_CG);
        ImGui::SetItemTooltip("The end colour replaces the brightest tones (highlights) in the image.");
        //The end colour replaces the brightest tones (highlights) in the image.
        //Start colour = what shadows become
        //End colour = what highlights become

        m_ColourGradBuffer.startColor = float3(startColor_CG[0], startColor_CG[1], startColor_CG[2]);
        m_ColourGradBuffer.endColor = float3(endColor_CG[0], endColor_CG[1], endColor_CG[2]);


        ImGui::TreePop();
}
ImGui::SetItemTooltip("Adjusts the colours and tones of an image to create a specific mood or aesthetic, enhancing visual storytelling through subtle shifts in hue, saturation, and contrast.");
```

```
 if (ImGui::TreeNode("Embossed")) {

        bool greyScale = m_EmbossBuffer.greyScale;
        ImGui::Checkbox("greyScale", &greyScale);
        m_EmbossBuffer.greyScale = greyScale;

        ImGui::SliderFloat("Embossed width", (float*)&m_EmbossBuffer.width_, 0.9, 1);
        ImGui::SetItemTooltip("How far along the x-axis the textures that make up the embosed effects are");

        ImGui::SliderFloat("Embossed height", (float*)&m_EmbossBuffer.height_, 0.9, 1);
        ImGui::SetItemTooltip("How far along the u-axis the textures that make up the embosed effects are");

        ImGui::TreePop();
 }
 ImGui::SetItemTooltip("Creates the illusion of raised or recessed surfaces on an image, adding depth and texture by highlighting edges with a contrast of light and shadow.");
```

```
if (ImGui::TreeNode("Glass Panel")) {

        m_GlassBuffer;

        ImGui::Checkbox("is pixle height and width linked", &panelinked);
        ImGui::SetItemTooltip("This makes the individual 'pannels' squares with the hight equaling the width");

        if (panelinked == true) {
                ImGui::SliderFloat("pannel size", (float*)&m_GlassBuffer.panelSize.x, 0, 20);
                ImGui::SetItemTooltip("The higher the number the smaller the square texture of the 'glass'.");
                m_GlassBuffer.panelSize.y = m_GlassBuffer.panelSize.x;
        }
        else {

                if (ImGui::TreeNode("Square Texture size  more")) {

                        ImGui::SliderFloat("pannel width", (float*)&m_GlassBuffer.panelSize.x, 0, 20);
                        ImGui::SetItemTooltip("How wide the individual pannel textures are");
                        ImGui::SliderFloat("pannel height", (float*)&m_GlassBuffer.panelSize.y, 0, 20);
                        ImGui::SetItemTooltip("How high/tall the individual pannel textures are");
                        ImGui::TreePop();
                }
        }

        ImGui::TreePop();
}
ImGui::SetItemTooltip("simulates the appearance of looking through a panel of squared textured glass");
```

```
        if (ImGui::TreeNode("Pixel")) {


                m_PixelBuffer.pixelSize;

                ImGui::Checkbox("Is pixle height and width linked", &pixelinked);
                ImGui::SetItemTooltip("This makes the individual pixels squares with the hight equaling the width");

                if (pixelinked == true) {
                        ImGui::SliderFloat("pixelSize", (float*)&m_PixelBuffer.pixelSize.x, 50, 200);
                        //ImGui::SetItemTooltip("the higher the number the more pixles are 'added'");
                        ImGui::SetItemTooltip("The higher the number the smaller the pixles become or the higher the 'resolution' becomes");
                        m_PixelBuffer.pixelSize.y = m_PixelBuffer.pixelSize.x;
                }
                else {

                        if (ImGui::TreeNode("pixelSize more")) {

                                ImGui::SliderFloat("pixel width", (float*)&m_PixelBuffer.pixelSize.x, 50, 200);
                                ImGui::SetItemTooltip("How wide the individual pixles are");
                                ImGui::SliderFloat("pixel height", (float*)&m_PixelBuffer.pixelSize.y, 50, 200);
                                ImGui::SetItemTooltip("How high/tall the individual pixles are");

                                ImGui::TreePop();
                        }
                }

                ImGui::TreePop();
        }
        ImGui::SetItemTooltip("Turns given image into a pixelated version by //");
}
```

///

```cpp
void PennyRender::PennyBoardIMGUI_savedEffectsTab()
{
	PennyBoardIMGUI_PreveiwScreen();

	ImGui::Separator();

	ImGui::SeparatorText("effect list");
	ImGui::SetItemTooltip("Where all saved / pre-built effects are stored");

	if (ImGui::TreeNode("pre - built effect list")) //not currently implemented
	{
		//ImGui::SeparatorText("pre-built effect list");

		ImGuiIO& io = ImGui::GetIO();
		ImTextureID my_tex_id = io.Fonts->TexID;

		float my_tex_w = (float)1366 / previewscreenSize;
		float my_tex_h = (float)768 / previewscreenSize;

		{
			static bool use_text_color_for_tint = false;

			ImVec2 uv_min = ImVec2(0.0f, 0.0f);                 // Top-left
			ImVec2 uv_max = ImVec2(1.0f, 1.0f);                 // Lower-right
			ImVec4 tint_col = use_text_color_for_tint ? ImGui::GetStyleColorVec4(ImGuiCol_Text) : ImVec4(1.0f, 1.0f, 1.0f, 1.0f);   // No tint
			ImVec4 border_col = ImGui::GetStyleColorVec4(ImGuiCol_Border);

			ImGui::PushID(0);
			{
				ImGui::Image(my_tex_id, ImVec2(my_tex_w / 2, my_tex_h / 2), uv_min, uv_max, tint_col, border_col);
				ImGui::SetItemTooltip("Image of the pre-built effect, to show the user a preveiw of said effect. ");

				ImGui::SameLine();
				ImGui::Text("Effect name \n" "list of effects active: \n - [] \n -[]");
				ImGui::SetItemTooltip("This is where the name of the effect is displayed,\n and currently show what effects were used to make this effect");

				ImGui::SameLine();
				ImGui::Button("Apply");
				ImGui::SetItemTooltip("This will apply your saved effect to the proper screen not the preview screen");

				ImGui::SameLine();
				if (ImGui::TreeNode("more info ")) {

					ImGui::SameLine();
					ImGui::Text("\n""info");

					ImGui::SameLine();
					ImGui::Text("\n""list of values: \n - [] \n -[]");

					ImGui::TreePop();
				}
				ImGui::SetItemTooltip("Show you a more INDEPTH reveiw of the effects and values used to produce the pre-built result");
			}
			ImGui::PopID();
```

```
                ImGui::PushID(1);
                {
                        ImGui::Image(my_tex_id, ImVec2(my_tex_w / 2, my_tex_h / 2), uv_min, uv_max, tint_col, border_col);
                        ImGui::SameLine();
                        ImGui::Text("Effect name \n" "list of effects active: \n - [] \n -[]");
                        ImGui::SameLine();
                        ImGui::Button("Apply ");

                        ImGui::SameLine();
                        if (ImGui::TreeNode("more info")) {

                                ImGui::SameLine();
                                ImGui::Text("\n""info");

                                ImGui::SameLine();
                                ImGui::Text("\n""list of values: \n - [] \n -[]");

                                ImGui::TreePop();
                        }
                }
                ImGui::PopID();

        }

        ImGui::TreePop();
}
ImGui::SetItemTooltip("Where all pre-built effects are stored");
```

```
        if (ImGui::TreeNode("user saved effect list")) // saved stuff isnt currently working
        {
                //ImGui::SeparatorText("user saved effect list");
                {
                        ImGui::Text("-Example Effect-");
                        ImGui::SetItemTooltip("This is where the name of the effect you have save is displayed, the name basically act like an ID for the effect");
                        ImGui::SameLine();
                        ImGui::Button("APPLY");
                        ImGui::SetItemTooltip("This will apply your saved effect to the proper screen not the preview screen");
                        //ImGui::SameLine();//add if i have time
                        //ImGui::Button("RENAME"); // make rename function
                        //ImGui::SetItemTooltip("This will bring a pop-up that allows you to rename your effect");
                        ImGui::SameLine();
                        ImGui::Button("DELETE"); // make delete function
                        ImGui::SetItemTooltip("This will delete currently stored effect, it is recoomended to only store X effects for memory and effecency sake");
                        ImGui::Text("- -");
                }

                //ImGui::SeparatorText("user saved effect list");
                if (savedEffect_List.size() != 0) {
                        for (int i = 0; i <= savedEffect_List.size(); i++)
                        {
                                ImGui::Text(savedEffect_List.data()->SavedName);
                                ImGui::SetItemTooltip("This is where the name of the effect you have save is displayed, the name basically act like an ID for the effect");
                                ImGui::SameLine();
                                ImGui::PushID(i); {

                                        if (ImGui::Button("APPLY")) {
                                                //ApplySavedEffect(savedEffect_List[i]);//this breaks everything - not sure why, come back later if time permits

                                        }
                                        ImGui::SetItemTooltip("This will apply your saved effect to the proper screen not the preview screen");
                                        //ImGui::SameLine();//add if i have time
                                        //ImGui::Button("RENAME"); // make rename function
                                        //ImGui::SetItemTooltip("This will bring a pop-up that allows you to rename your effect");
                                        ImGui::SameLine();
                                        if (ImGui::Button("DELETE")) {
                                                removeEffect(savedEffect_List, i);
                                        }
                                        ImGui::SetItemTooltip("This will delete currently stored effect, it is recoomended to only store X effects for memory and effecency sake");
                                        ImGui::Text("- -");


                                        ImGui::PopID();
                                }
                        }

                }


                ImGui::TreePop();
        }
        ImGui::SetItemTooltip("Where all user saved/made effects are stored");
}
```

```
void PennyRender::PennyBoardIMGUI_toolCustomTab()
{

        ImGui::ShowStyleEditor();

}
```

```
void PennyRender::PennyBoardIMGUI() {

	ImGui::Begin("Penny_board Graphics (shader toggle tool)");//creates new window

	ImGui::Separator();
	if (ImGui::BeginTabBar("##tabs", ImGuiTabBarFlags_None))
	{
		if (ImGui::BeginTabItem("Editor")) {

			PennyBoardIMGUI_editorTab();

			ImGui::EndTabItem();

		}
		if (ImGui::BeginTabItem("Saved Effects list")) {

			PennyBoardIMGUI_savedEffectsTab();

			ImGui::EndTabItem();

		}
		if (ImGui::BeginTabItem("Tool Cus")) {

			PennyBoardIMGUI_toolCustomTab();

			ImGui::EndTabItem();

		}

		ImGui::EndTabBar();
	}

	//ImGui::Separator();

	//ImGui::ShowDemoWindow();

	//ImGui::ShowStyleEditor();

	//ImGui::ShowStyleEditor(ImGuiStyle* ref);

	ImGui::End();//ends that window

}
```

Renderer Code

Popping open the engine/hood // under the hood

```
#pragma once

//#include "Skateboard/Renderer/GraphicsContext.h"
#include "Skateboard/Graphics/RHI/ResourceFactory.h"
#include "Skateboard/Graphics/RHI/RenderCommand.h"
#include "Skateboard/Renderers/Renderer.h"

//#include "Penny-Board/TestRenderer.h"
using namespace Skateboard;

namespace pen
{
```

Buffer stuff

Buffer Struct

```
struct ScreenBuffer //ScreenPara            //b2
{
float2 screenSize_;
float2 padding; //edit later
//ScreenBuffer() { padding = float2(1200, 800); screenSize_ = float2(1200, 800);}
ScreenBuffer() { padding = float2(1.f, 1.f); screenSize_ = float2(1366, 768); }
};
```

Screen buffer passes through the screen size to any potential shader if initialised

```
struct ScanlineBuffer//scanlinePara //b3 { float2 screenSize_; float LineThickness_; float DimmedFactor_;

   float transferPower_;
    float vertical_;
    int PushData_instanceNo;
    float padding;

    ScanlineBuffer() { PushData_instanceNo = 0; screenSize_ = float2(1366, 768); LineThickness_ = 3.f;
DimmedFactor_ = 0.5; transferPower_ = 0; vertical_ = 0.5; }

    ScanlineBuffer(float linethickness) :
        ScanlineBuffer() {
        LineThickness_ = linethickness;
    }
    ScanlineBuffer(float linethickness, float dimmedFactor) :
        ScanlineBuffer(linethickness) {
        DimmedFactor_ = dimmedFactor;
    }
    ScanlineBuffer(float linethickness, float dimmedFactor, float transferPower) :
        ScanlineBuffer(linethickness, dimmedFactor) {
        transferPower_ = transferPower;
    }
    ScanlineBuffer(float linethickness, float dimmedFactor, float transferPower, float vertical) :
        ScanlineBuffer(linethickness, dimmedFactor, transferPower) {
        vertical_ = vertical;
    }
```

```
};
```

```
struct VignettingBuffer//vignettingPara //b4 { float2 screenSize_; float2 padding;

   float innerRadius;
    float outerRadius;
    float opacity;
    int PushData_instanceNo;

    VignettingBuffer() { PushData_instanceNo = 0; screenSize_ = float2(1366, 768); innerRadius = 0.f;
outerRadius = 0.4; opacity = 0.8f; padding = float2(0,0); }
};
```

```
struct PosterizeBuffer//posterizePara        //b5
{
    float3 step_areas;
    int PushData_instanceNo;
    PosterizeBuffer() { PushData_instanceNo = 0; step_areas = float3(3.8f, 1.4f, 0.2f); }
};
```

```
struct BloomBuffer //bloomPara
{
int PushData_instanceNo;
int width;
float angleSteps;
float radiusSteps;

float ampFactor;
float3 padding;

BloomBuffer() { PushData_instanceNo = 0; width = 20; angleSteps = 20; radiusSteps = 20; ampFactor = 1.25; }
};
```

```
struct Bloom2Buffer //bloom2Para
{
int PushData_instanceNo;
float ampFactor;
float threshold;
float padding0;

Bloom2Buffer() { PushData_instanceNo = 0; ampFactor = 1; threshold = 0.24; padding0 = 0; }
};
```

```
struct BlurGauBuffer //blurGauPara  //
{
float Luminance;
float2 screenSize_;
float padding0;

//float weight[5];
//float padding1[3]; // padding to align next array
```

```
//float texscrMultiplier[5];
//float padding2[3]; // padding for alignment

float weight[8]; //last values are padding
float texscrMultiplier[8];//last values ar padding


BlurGauBuffer() {
for (int i = 0; i < 8; i++) { weight[i] = 0.2; float placeholder = i; texscrMultiplier[i] = placeholder /
4; }
padding0 = 0; screenSize_ = float2(1366, 768); Luminance = 0.4;
}
};
```

```
struct ChromaTrueBuffer //ChromaTruePara { int PushData_instanceNo; float2 screenSize_; float
aberrationFactor;

   float aberrationFactor_x;
    float aberrationFactor_y;
    float2 padding; // padding for alignment

    ChromaTrueBuffer() { PushData_instanceNo = 0; screenSize_ = float2(1366, 768); padding = float2(0,0);
aberrationFactor = 0.009; aberrationFactor_x = 0.5; aberrationFactor_y = 0.5; }
};
```

```
struct ChromaStylizedBuffer //ChromaStylizedPara
{                                                    //might not need arrays - matters how customizable i want
it

int PushData_instanceNo;
float3 padding0;

float4 chromaticWeights1;
float4 chromaticWeights2;
float4 chromaticOffset1;
float4 chromaticOffset2;

ChromaStylizedBuffer() {

//for (int i = 0; i < 1; i++)
{
chromaticOffset1 = float4(0.01, 0.01, 13, 0);
chromaticOffset2 = float4(-0.01, -0.01, 13, 0);
chromaticWeights1 = float4(0.2, 0.2f, 0.f, 0.25);
chromaticWeights2 = float4(0.2f, 0.f, 0.2, 0.25);
}
padding0 = float3(0,0,0);
 PushData_instanceNo =0;
}
};
```

```
struct ColourGradBuffer //colourGradPara
{
int PushData_instanceNo;
float3 padding0;

float3 startColor;
float padding1;
```

```
float3 endColor;
float padding2;


ColourGradBuffer() { startColor = float3(1, 1, 1); endColor = float3(1.2, 0.01, 0.51); PushData_instanceNo =
0;
}
};
```

```
struct EmbossBuffer //EmbossPara {

   int PushData_instanceNo;
    float2 screenSize_;
    float padding0;

    int greyScale;
    float width_;
    float height_;
    float padding1;

    EmbossBuffer() { PushData_instanceNo = 0; screenSize_ = float2(1366, 768); greyScale = true; width_ =
400; height_ = 400; padding0 = 0; padding1 = 0; }
};
```

```
struct GlassBuffer //glassPara { int PushData_instanceNo; float2 screenSize_; float2 panelSize;

   float2 padding; // for alignment

    GlassBuffer() { PushData_instanceNo = 0; screenSize_ = float2(1366, 768); panelSize = float2(10, 10);
padding = float2(10, 10); }
};
```

```
struct PixelBuffer //pixelPara { int PushData_instanceNo; float2 screenSize_; float2 pixelSize;

   float2 padding; // for alignment

    PixelBuffer() { PushData_instanceNo = 0; screenSize_ = float2(1366, 768); pixelSize = float2(200, 200);
padding = float2(200, 200); }
};
```

```
struct SavedEffect {
//string SavedName;
const char* SavedName;
ShaderToggles EffectInUse;

ScreenBuffer screen_buff;
ScanlineBuffer scanline_buff;
VignettingBuffer vignet_buff;
PosterizeBuffer posterize_buff;
BloomBuffer bloom_buff;
```

```
Bloom2Buffer bloom2_buff;
BlurGauBuffer blurGauBuffer_buff;
ChromaTrueBuffer  chromaTrue_buff;
ChromaStylizedBuffer chromaStylized_buff;
ColourGradBuffer colourGrad_buff;
EmbossBuffer emboss_buff;
GlassBuffer glass_buff;
PixelBuffer pixel_buff;

};
```

```
struct Vertex
{
float3 Position;
float3 Colour;
float2 UV;
float3 Normal;

//default colour is white
Vertex() : Position(0, 0, 0), Colour(0, 0, 0), UV(0, 0), Normal(0, 0, 0) {}
Vertex(float3 pos) : Vertex() { Position = pos; }
Vertex(float3 pos, float3 col) : Vertex(pos) { Colour = col; }
Vertex(float3 pos, float3 col, float2 uv) : Vertex(pos, col) { UV = uv; }
Vertex(float3 pos, float3 col, float2 uv, float3 normal) : Vertex(pos, col, uv) { Normal = normal; }

static BufferLayout VertexLayout()
{
return {
{ POSITION,     ShaderDataType_::Float3 },
{ COLOUR,       ShaderDataType_::Float3 },
{ TEXCOORD,     ShaderDataType_::Float2 },
{ NORMAL,       ShaderDataType_::Float3 },
};
}
};
```

```
enum LightType : uint32_t
{
LightDirectional = 0,
LightPoint = 1,
LightSpot = 2,
};
```

```
struct InstanceData
{
matrix World;
float4 ColourScale;

int TextureIndex;

float4 SpecularColor;
float SpecularPower;
float SpecularWeight;

InstanceData() { World = glm::identity<glm::mat4x4>(); ColourScale = float4(1, 1, 1, 1); SpecularColor = float4(1, 1, 1,
1);  TextureIndex = 1; SpecularPower = 1.f; SpecularWeight = 1; }
};
```

```
struct Light
{
public:
```

```
float4 DiffuseColour;
float4 Attenuation;

float3 LightPosition;
float InnerCone;

float3 LightDirection;
float OuterCone;

private:
float2 Padding;
public:
float FalloffPower;
uint32_t LightType;

static float4 AttenuationDefaults() { return { 0.05f, 0.01f, 0.001f, 100.f }; }
};
```

```
struct  FrameData
{
glm::mat4x4 ViewMatrix;
glm::mat4x4 ProjectionMatrix;
};
```

```
struct Frame  //change to be more like screen fram in raytracer
{
FrameData Matrices;

matrix CameraMatrix;
uint32_t LightCount;
float3 AmbientLight;
};
```

```
static SamplerDesc AnisotropicSampler(uint8_t anisotropy, SamplerMode_ U, SamplerMode_ V)
{
    return
    {
        .Filter = SamplerFilter_::SamplerFilter_Anisotropic,
        .ModeU = U,
        .ModeV = V,
        .ModeW = V,
        .MipMapLevelOffset = 0.f,
        .MipMapMinSampleLevel = 0.f,
        .MipMapMaxSampleLevel = 10.f,
        .MaxAnisotropy = anisotropy,    // Valid range 1 - 16 -> uint32_t cause padding anyways
        .ComparisonFunction = SamplerComparisonFunction_Less_Equal,
        .BorderColour = SamplerBorderColour_TransparentBlack,
        .Flags = 0,
    };
}
```

```
static SamplerDesc LinearSampler(SamplerMode_ U, SamplerMode_ V)
{
    auto sampler = SamplerDesc::InitAsDefaultTextureSampler();
    sampler.ModeU = U;
    sampler.ModeV = V;
    return sampler;
}
```

```
static SamplerDesc PointSampler(SamplerMode_ U, SamplerMode_ V)
{
    auto sampler = SamplerDesc::InitAsDefaultTextureSampler();
    sampler.Filter = SamplerFilter_::SamplerFilter_Comaprison_Min_Mag_Mip_Point;
```

```
    sampler.ModeU = U;
    sampler.ModeV = V;
    return sampler;
}
```

```
//post-processing pipelines + "toggles"
enum ShaderToggles : uint16_t
{
    //-- post-proccessing effects // shaders
    PIXELATE_ = 0,
    POSTERIZE_ = 1,
    SCANLINE_ = 2,
    VIGNETTE_ = 3,


    BLOOM_ = 4,
    CHROMABER_TRUE_ = 5,
    CHROMABER_STYLE_ = 6,


    COLOURGRAD_ = 7,
    GLASS_ = 8,
    EMBOS_ = 9,
    BLUR_GAU_ = 10,
    POSTPROSSESSING_ = 11
    //TEST_ = 1 << 10
    //--
};
```

```
ENUM_FLAG_OPERATORS(ShaderToggles);
```

```
constexpr static ShaderToggles DEFAULT_STATE_ = POSTPROSSESSING_;
constexpr static uint16_t EFFECT_PERMUTATIONS = 12U;//(PIXELATE_ | POSTERIZE_ | SCANLINE_ | VIGNETTE_ | BLOOM_ |
CHROMABER_TRUE_ | CHROMABER_STYLE_ | COLOURGRAD_ | GLASS_ | EMBOS_ | BLUR_GAU_ | POSTPROSSESSING_) + 1;
```

Setting up the buffers

```
//normal rendering

    Skateboard::PipelineRef NormalVisualizer;

    size_t DynamicBufferSize = 64 * 1024;
    size_t InstanceDataBufferSize = 64 * 1024;


    Skateboard::MultiResource<Skateboard::BufferRef> TriangleDataBuffer;
    Skateboard::MultiResource<Skateboard::BufferRef> InstanceDataBuffer;
    Skateboard::MultiResource<Skateboard::BufferRef> LightDataBuffer;


    //--------------
    // post-processing buffers
```

```
    size_t pennyBufferSize = 64 * 1024; //

    //Buffer refs
    Skateboard::MultiResource<Skateboard::BufferRef> ScreenDataBuffer;
    Skateboard::MultiResource<Skateboard::BufferRef> ScanlineDataBuffer;
    Skateboard::MultiResource<Skateboard::BufferRef> VignettingDataBuffer;
    Skateboard::MultiResource<Skateboard::BufferRef> PosterizeDataBuffer;


    Skateboard::MultiResource<Skateboard::BufferRef> BloomDataBuffer;
    Skateboard::MultiResource<Skateboard::BufferRef> Bloom2DataBuffer;
    Skateboard::MultiResource<Skateboard::BufferRef> BlurGauDataBuffer;
    Skateboard::MultiResource<Skateboard::BufferRef> ChromaTrueDataBuffer;


    Skateboard::MultiResource<Skateboard::BufferRef> ChromaStylizedDataBuffer;
    Skateboard::MultiResource<Skateboard::BufferRef> ColourGradDataBuffer;
    Skateboard::MultiResource<Skateboard::BufferRef> EmbossDataBuffer;
    Skateboard::MultiResource<Skateboard::BufferRef> GlassDataBuffer;


    Skateboard::MultiResource<Skateboard::BufferRef> pixelDataBuffer;


    // desc
    BufferViewDesc screen_desc;
    BufferViewDesc scanline_desc;
    BufferViewDesc vignetting_desc;
    BufferViewDesc posterize_desc;


    BufferViewDesc bloom_desc;
    BufferViewDesc bloom2_desc;
    BufferViewDesc blurGau_desc;
    BufferViewDesc ChromaTrue_desc;


    BufferViewDesc ChromaStylized_desc;
    BufferViewDesc colourGrad_desc;
    BufferViewDesc Emboss_desc;
    BufferViewDesc glass_desc;


    BufferViewDesc pixel_desc;


    // init buffer structs
    ScreenBuffer m_ScreenBuffer = ScreenBuffer();
    ScanlineBuffer m_ScanlineBuffer = ScanlineBuffer();
    VignettingBuffer m_VignettingBuffer = VignettingBuffer();
    PosterizeBuffer m_PosterizeBuffer = PosterizeBuffer();


    BloomBuffer    m_BloomBuffer = BloomBuffer();
    Bloom2Buffer    m_Bloom2Buffer = Bloom2Buffer();
    BlurGauBuffer m_BlurGauBuffer = BlurGauBuffer();
    ChromaTrueBuffer m_ChromaTrueBuffer = ChromaTrueBuffer();


    ChromaStylizedBuffer m_ChromaStylizedBuffer = ChromaStylizedBuffer();
    ColourGradBuffer m_ColourGradBuffer = ColourGradBuffer();
    EmbossBuffer m_EmbossBuffer = EmbossBuffer();
    GlassBuffer m_GlassBuffer = GlassBuffer();


    PixelBuffer m_PixelBuffer = PixelBuffer();


    // Pipeline default sate inti
    ShaderToggles CURRENT_STATE_ = DEFAULT_STATE_;//
    //ShaderToggles CURRENT_STATE_Pass[EFFECT_PERMUTATIONS];// DEPTH_TEST;
    //ShaderToggles CURRENT_STATE_Pass[5] =
{ DEFAULT_STATE_ ,DEFAULT_STATE_ ,DEFAULT_STATE_ ,DEFAULT_STATE_ ,DEFAULT_STATE_ };


    //--------------
    //inline view desc
    BufferViewDesc sbvdesc;
    BufferViewDesc lightsbvdesc;
    BufferViewDesc cbvdesc;
```

```
    //resterizer
    SamplerDesc StaticSamplerDesc = SamplerDesc::InitAsDefaultTextureSampler();


    //counts offsets for dynamic data uploaded every frame
    size_t m_Offset = ROUND_UP(sizeof(FrameData), GraphicsConstants::CONSTANT_BUFFER_ALIGNMENT);


    //forked on each call with a unique data pointer/ otherwise default instance data is used
    uint32_t m_InstanceDataForks = 0;


    Frame m_Frame{ .AmbientLight = { 0.1, 0.1, 0.1} };
    FrameData m_CameraData{};


    std::vector<Light> m_Lights;


    InstanceData m_DefaultInstanceData = InstanceData();


    uint32_t m_DefaultTextureIDX = 0;


    bool m_Pipeline_dirty;
    bool m_VisualiseNormals = false;// probs delete
```

Rendering & Post-Processing Pipeline Variables Documentation
This section documents the state and buffer declarations for a post-processing rendering pipeline built with the Skateboard engine framework. It includes buffers for scene rendering, post-processing effects, light data, and dynamic frame updates.

Normal Rendering Pipeline
```
Skateboard::PipelineRef NormalVisualizer;
```

Purpose: Pipeline reference for rendering scene geometry with standard or visual debug shaders (e.g., normal visualizations).

Dynamic & Instance Buffers
```
size_t DynamicBufferSize = 64 * 1024;
size_t InstanceDataBufferSize = 64 * 1024;
```

DynamicBufferSize: Memory allocated for frame-dependent dynamic data (e.g., camera info, light count, etc.).
InstanceDataBufferSize: Storage for per-instance data such as transform matrices or material properties.

Scene Buffers
```
Skateboard::MultiResource<Skateboard::BufferRef> TriangleDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> InstanceDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> LightDataBuffer;
```

TriangleDataBuffer: Stores vertex/index data or triangle-level metadata for rendering.
InstanceDataBuffer: Holds data for each instance rendered this frame.
LightDataBuffer: Contains information on dynamic and static lights (position, color, intensity).

Post-Processing Buffers
```
size_t pennyBufferSize = 64 * 1024;
```

A general-purpose allocation size used for each post-processing buffer

ach effect has its own BufferRef stored in a MultiResource (multi-frame resource safe for double/triple buffering):

```
Skateboard::MultiResource<Skateboard::BufferRef> ScreenDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> ScanlineDataBuffer;
```

```
Skateboard::MultiResource<Skateboard::BufferRef> VignettingDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> PosterizeDataBuffer;

//Bloom and blur type effects
Skateboard::MultiResource<Skateboard::BufferRef> BloomDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> Bloom2DataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> BlurGauDataBuffer;


// Chromatic effects
Skateboard::MultiResource<Skateboard::BufferRef> ChromaTrueDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> ChromaStylizedDataBuffer;


Skateboard::MultiResource<Skateboard::BufferRef> ColourGradDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> EmbossDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> GlassDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> pixelDataBuffer;

-------------------------------------------------------------------------------------------------------------------
Buffer View Descriptions
BufferViewDesc screen_desc;
BufferViewDesc scanline_desc;
...
BufferViewDesc pixel_desc;

Purpose: Describe how each buffer is viewed by the GPU (CBV, SRV, UAV, etc.).
Each effect has a corresponding descriptor that links the CPU-side buffer to a GPU-side resource binding.


-------------------------------------------------------------------------------------------------------------------
Buffer Struct Instances
ScreenBuffer m_ScreenBuffer;
ScanlineBuffer m_ScanlineBuffer;
...
PixelBuffer m_PixelBuffer;

These structs hold initialized per-effect data, ready to be copied to their corresponding GPU buffer before a draw or
dispatch.
Initialized once here, but typically updated each frame with relevant scene-dependent values (resolution, animation,
toggles).


-------------------------------------------------------------------------------------------------------------------
Pipeline State Tracking
ShaderToggles CURRENT_STATE_ = DEFAULT_STATE_;

Tracks the active shader features or combinations (toggle flags).
Could include toggles like USE_VIGNETTE, ENABLE_BLOOM, or VISUALIZE_NORMALS.


-------------------------------------------------------------------------------------------------------------------
Global Buffer Descriptors
BufferViewDesc sbvdesc;       // Screen-space descriptor (scene buffer)
BufferViewDesc lightsbvdesc;  // Light buffer descriptor
BufferViewDesc cbvdesc;       // Constant buffer descriptor (camera/frame)

Purpose: Low-level descriptor used when binding data to GPU pipeline.
Abstracts away root signature or descriptor table details.


-------------------------------------------------------------------------------------------------------------------
Sampler State
SamplerDesc StaticSamplerDesc = SamplerDesc::InitAsDefaultTextureSampler();

Default sampler used in pixel shaders.
Typically includes linear filtering and clamp addressing.

-------------------------------------------------------------------------------------------------------------------
Frame-Dependent Data
size_t m_Offset = ROUND_UP(sizeof(FrameData), GraphicsConstants::CONSTANT_BUFFER_ALIGNMENT);

Computes aligned offset for frame-constant buffers.
Ensures safe and efficient memory writes per-frame.

-------------------------------------------------------------------------------------------------------------------
Instance Forking (Instancing)
```

```
uint32_t m_InstanceDataForks = 0;

Tracks the number of times instance data has been duplicated this frame (e.g., for branching visual states, special
effects).

-----------------------------------------------------------------------------------------------------------------------
Scene Lighting and Frame Metadata
Frame m_Frame{ .AmbientLight = { 0.1, 0.1, 0.1} };
FrameData m_CameraData{};
std::vector<Light> m_Lights;
InstanceData m_DefaultInstanceData = InstanceData();

m_Frame: Global frame constants, including ambient lighting.
m_CameraData: Per-frame camera matrices and frustum data.
m_Lights: Collection of point/spot/directional lights in the scene.
m_DefaultInstanceData: Used when no override is supplied during rendering.

-----------------------------------------------------------------------------------------------------------------------
Texture Binding Default
uint32_t m_DefaultTextureIDX = 0;

Fallback or default texture index used when no texture is explicitly bound to a material or instance.

-----------------------------------------------------------------------------------------------------------------------
Render State Dirty Flag
bool m_Pipeline_dirty;

If true, the rendering pipeline or shader bindings need to be recompiled or rebound before the next frame.

-----------------------------------------------------------------------------------------------------------------------
Debug Toggle for Normals

bool m_VisualiseNormals = false;

When enabled, shaders visualize surface normals—useful for debugging lighting or tangent-space calculations.
```

```
 void InitBuffers(BufferDesc bufferdesc) {


        //ScreenDataBuffer Buffer
        bufferdesc.Init(pennyBufferSize, ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead);
        ScreenDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer(bufferdesc); });


        //ScanlineDataBuffer Buffer
        ScanlineDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({ .AccessFlags =
ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(ScanlineBuffer) }); });


        //VignettingDataBuffer Buffer
        VignettingDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({ .AccessFlags =
ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(VignettingBuffer) }); });


        //PosterizeDataBuffer Buffer
        PosterizeDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({.AccessFlags =
ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(PosterizeBuffer)}); });


        BloomDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({ .AccessFlags =
ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(BloomBuffer) }); });
        Bloom2DataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({ .AccessFlags =
ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(Bloom2Buffer) }); });
        BlurGauDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({ .AccessFlags =
ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(BlurGauBuffer) }); });
        ChromaTrueDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({ .AccessFlags =
ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(ChromaTrueBuffer) }); });
        ChromaStylizedDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({ .AccessFlags =
ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(ChromaStylizedBuffer) }); });
        ColourGradDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({ .AccessFlags =
ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(ColourGradBuffer) }); });
        EmbossDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({ .AccessFlags =
ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(EmbossBuffer) }); });
        GlassDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({ .AccessFlags =
ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(GlassBuffer) }); });
```

```
        pixelDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer({ .AccessFlags =
ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead, .Size = sizeof(PixelBuffer) }); });


    }
```

InitBuffers(BufferDesc bufferdesc) — Buffer Initialization Function

---

Purpose

This function allocates and initializes all CPU-writeable / GPU-readable buffers required by the post-processing system.
Each buffer corresponds to a specific effect like scanlines, bloom, chromatic aberration, emboss, etc.
These buffers will later be updated with their respective data structs (like ScanlineBuffer, BloomBuffer, etc.) before being uploaded to the GPU.

Parameters

BufferDesc bufferdesc

A reusable buffer description object passed in from outside.
The Init method is used to set:
Size: e.g. pennyBufferSize (64 KB default)
AccessFlags: CPU write access and GPU read access, suitable for constant buffers.

Function Body Walkthrough

```
// ScreenDataBuffer
bufferdesc.Init(pennyBufferSize, ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead);
ScreenDataBuffer.ForEach([&](BufferRef& ref) {
    ref = ResourceFactory::CreateBuffer(bufferdesc);
});
```

Buffer: ScreenDataBuffer
Usage: Holds general screen-space rendering info like screen resolution.
Access: CPU-writable for updates each frame, GPU-readable in shaders.

Per-Effect Buffers

Each of the following follows this template:

```
Buffer.ForEach([&](BufferRef& ref) {
    ref = ResourceFactory::CreateBuffer({
        .AccessFlags = ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead,
        .Size = sizeof(CorrespondingBufferStruct)
    });
});
```

List of Buffers and Their Purpose:

Buffer	Struct	Purpose
ScanlineDataBuffer	ScanlineBuffer	CRT scanline simulation
VignettingDataBuffer	VignettingBuffer	Darkens screen edges to focus viewer's attention
PosterizeDataBuffer	PosterizeBuffer	Color quantization for stylized/retro effects
BloomDataBuffer	BloomBuffer	Radial light bleeding (intense light bloom)
Bloom2DataBuffer	Bloom2Buffer	Simpler threshold-based bloom effect
BlurGauDataBuffer	BlurGauBuffer	Configurable Gaussian blur filter
ChromaTrueDataBuffer	ChromaTrueBuffer	Basic chromatic aberration (RGB channel offset)
ChromaStylizedDataBuffer	ChromaStylizedBuffer	Advanced multi-channel chromatic distortions
ColourGradDataBuffer	ColourGradBuffer	Gradient overlays between two colors
EmbossDataBuffer	EmbossBuffer	Screen-space bump/edge highlighting for stylized visuals
GlassDataBuffer	GlassBuffer	Simulates refraction through glass panels or water
pixelDataBuffer	PixelBuffer	Low-resolution pixelation filter

Implementation Notes

MultiResource Pattern: Each buffer is wrapped in a MultiResource object, enabling use in multi-frame rendering systems (e.g., triple buffering or per-frame resource sets).
Access Flags: All buffers are marked with:

CpuWrite: For CPU-side updates each frame.
GpuRead: Allows shaders to access the data in rendering or post-processing passes.
Size Calculation: For most buffers, sizeof(Struct) ensures only the required space is reserved.

```
void InitData(uint32_t InstanceDataBufferSize) {

int offset = 0; //nothing to offset by /-change cmment layter

//ScreenDATA
screen_desc.InitAsStructuredBuffer<ScreenBuffer>(offset, InstanceDataBufferSize / sizeof(ScreenBuffer));

//ScanlineDATA
scanline_desc.InitAsStructuredBuffer<ScanlineBuffer>(offset, InstanceDataBufferSize / sizeof(ScanlineBuffer));

//vignettingDATA
vignetting_desc.InitAsStructuredBuffer<VignettingBuffer>(offset, InstanceDataBufferSize / sizeof(VignettingBuffer));

//posterizeDATA
posterize_desc.InitAsStructuredBuffer<PosterizeBuffer>(offset, InstanceDataBufferSize / sizeof(PosterizeBuffer));

//bloomDATA
bloom_desc.InitAsStructuredBuffer<BloomBuffer>(offset, InstanceDataBufferSize / sizeof(BloomBuffer));

//bloom2DATA
bloom2_desc.InitAsStructuredBuffer<Bloom2Buffer>(offset, InstanceDataBufferSize / sizeof(Bloom2Buffer));

//blurGauDATA
blurGau_desc.InitAsStructuredBuffer<BlurGauBuffer>(offset, InstanceDataBufferSize / sizeof(BlurGauBuffer));

//ChromaTrueDATA
ChromaTrue_desc.InitAsStructuredBuffer<ChromaTrueBuffer>(offset, InstanceDataBufferSize / sizeof(ChromaTrueBuffer));

//ChromaStylizedDATA
ChromaStylized_desc.InitAsStructuredBuffer<ChromaStylizedBuffer>(offset, InstanceDataBufferSize /
sizeof(ChromaStylizedBuffer));

//colourGradDATA
colourGrad_desc.InitAsStructuredBuffer<ColourGradBuffer>(offset, InstanceDataBufferSize / sizeof(ColourGradBuffer));

//embossDATA
Emboss_desc.InitAsStructuredBuffer<EmbossBuffer>(offset, InstanceDataBufferSize / sizeof(EmbossBuffer));

//glassDATA
glass_desc.InitAsStructuredBuffer<GlassBuffer>(offset, InstanceDataBufferSize / sizeof(GlassBuffer));

//pixelDATA
pixel_desc.InitAsStructuredBuffer<PixelBuffer>(offset, InstanceDataBufferSize / sizeof(PixelBuffer));

}
```

InitData(uint32_t InstanceDataBufferSize) — Descriptor Initialization Function

---

Purpose

This function initializes GPU buffer view descriptors (of type BufferViewDesc) for each post-processing effect buffer.
These descriptors define how shaders read structured data (e.g., uniform-like structures) from memory.
Each view maps a specific type (e.g., ScanlineBuffer, BloomBuffer) to a region of a buffer, enabling structured access on the GPU side.

Parameters

uint32_t InstanceDataBufferSize

Total memory size available for structured data (in bytes).
Used to compute the number of struct instances per buffer.
Ensures consistent allocation across different buffers.

Function Body Walkthrough

int offset = 0; // nothing to offset by / - change comment later

Offset into the buffer, currently set to 0 for all views (no subregioning yet).
Placeholder for future support for buffer suballocations (i.e., offset-based binding).

Descriptor Initialization Pattern

Each line follows the same structure:
```
descriptor.InitAsStructuredBuffer<StructType>(
    offset,
    InstanceDataBufferSize / sizeof(StructType)
);
```

StructType: The buffer structure to be read by the GPU (e.g., PosterizeBuffer).
offset: Offset into the buffer (currently always 0).
count: Number of struct elements that can fit in the allocated buffer.

Post-Processing Buffer View Descriptors

Descriptor	Struct Type	Purpose
screen_desc	ScreenBuffer	Holds screen dimensions and padding
scanline_desc	ScanlineBuffer	Defines scanline strength, direction, dimming
vignetting_desc	VignettingBuffer	Holds inner/outer radius and opacity for vignette effect
posterize_desc	PosterizeBuffer	Describes color step sizes for posterization
bloom_desc	BloomBuffer	Describes bloom radii and amplification
bloom2_desc	Bloom2Buffer	Alternate bloom method using threshold/amp
blurGau_desc	BlurGauBuffer	Gaussian blur parameters and sampling weights
ChromaTrue_desc	ChromaTrueBuffer	Simple chromatic aberration offsets
ChromaStylized_desc	ChromaStylizedBuffer	Advanced chromatic offsets and color weighting
colourGrad_desc	ColourGradBuffer	Defines gradient transition between two colors
Emboss_desc	EmbossBuffer	Emboss width, height, and greyscale toggle
glass_desc	GlassBuffer	Glass pane size, used for distortion effects
pixel_desc	PixelBuffer	Pixel size data for pixelation effect

Why This Step Matters

These descriptors are what allow the GPU to see the buffer contents correctly.
Without this, data may be interpreted incorrectly or misaligned.
The structured format also gives type safety on the shader side, aligning with constant buffer layout rules.

```
 void BeginIncrementCounter() {
        //m_ScanlineBuffer = ScanlineBuffer(LineThickness_, DimmedFactor_, transferPower_, vertical_);
        ScreenDataBuffer.IncrementCounter();
        ScanlineDataBuffer.IncrementCounter();
        VignettingDataBuffer.IncrementCounter();
        PosterizeDataBuffer.IncrementCounter();


        BloomDataBuffer.IncrementCounter();
        Bloom2DataBuffer.IncrementCounter();
        BlurGauDataBuffer.IncrementCounter();
        ChromaTrueDataBuffer.IncrementCounter();


        ChromaStylizedDataBuffer.IncrementCounter();
        ColourGradDataBuffer.IncrementCounter();
        EmbossDataBuffer.IncrementCounter();
        GlassDataBuffer.IncrementCounter();


        pixelDataBuffer.IncrementCounter();


    }
```

BeginIncrementCounter() — Advance Frame Resource Counters

Purpose

This function increments the internal resource counters for all post-processing MultiResource buffers, enabling them to switch to the next buffer instance for the current frame. It’s designed to be called once per frame (typically at the start of the frame update or rendering pass).

Context: MultiResource Buffering

MultiResource<T> is a wrapper that contains a buffer of GPU buffer references.
Internally, IncrementCounter() moves to the next available buffer slot.

Allowing the CPU to prepare data while the GPU is still reading previous frame resources.

Affected Buffers

The following buffers have their internal index advanced:
Buffer, Purpose / Contents
ScreenDataBuffer, Contains screen resolution and padding
ScanlineDataBuffer, Controls scanline rendering parameters
VignettingDataBuffer, Stores vignette falloff and opacity data
PosterizeDataBuffer, Contains parameters for posterization effect
BloomDataBuffer, Parameters for radial bloom/glow
Bloom2DataBuffer, Alternate bloom with amp/threshold settings
BlurGauDataBuffer, Gaussian blur weight and sampling configuration
ChromaTrueDataBuffer, Basic chromatic aberration settings
ChromaStylizedDataBuffer, Advanced chromatic offset/weight combinations
ColourGradDataBuffer, Gradient colour data (start/end)
EmbossDataBuffer, Emboss intensity, greyscale toggle, dimensions
GlassDataBuffer, Simulates panel-like refraction or distortion
pixelDataBuffer, Pixelation strength and pixel block size

Implementation Notes

Safe Resource Management: Ensures each frame’s data write doesn’t overwrite data in use by the GPU.
Temporal Effects: If any of these buffers support motion-aware or frame-history-based effects, this system enables it.
Expandability: Adding new buffers to this system simply requires calling IncrementCounter() on the new MultiResource.

```
  void EndCopyToBuffer() {

        int offset = 0;

            //GraphicsContext::CopyDataToBuffer(PosterizeDataBuffer.Get().get(), offset, sizeof(Frame), &m_Frame);
            //GraphicsContext::CopyDataToBuffer(PosterizeDataBuffer.Get().get(), offset, sizeof(PosterizeBuffer),
&m_PosterizeBuffer);



            //switch (EFFECT_PERMUTATIONS)
            switch (CURRENT_STATE_)
            //switch (ShaderToggles)
            {
            case PIXELATE_:
                GraphicsContext::CopyDataToBuffer(pixelDataBuffer.Get().get(), offset, sizeof(PixelBuffer),
&m_PixelBuffer);
                break;
            case POSTERIZE_:
                GraphicsContext::CopyDataToBuffer(PosterizeDataBuffer.Get().get(), offset, sizeof(PosterizeBuffer),
&m_PosterizeBuffer);
                break;
            case SCANLINE_:
                GraphicsContext::CopyDataToBuffer(ScanlineDataBuffer.Get().get(), offset, sizeof(ScanlineBuffer),
&m_ScanlineBuffer);
                break;
            case VIGNETTE_:
                GraphicsContext::CopyDataToBuffer(VignettingDataBuffer.Get().get(), offset, sizeof(VignettingBuffer),
&m_VignettingBuffer);
                break;
            case BLOOM_:
                //RSPD.SetPixelShader(L"bloom_ps");
                GraphicsContext::CopyDataToBuffer(Bloom2DataBuffer.Get().get(), offset, sizeof(Bloom2Buffer),
&m_Bloom2Buffer);
                break;
            case CHROMABER_TRUE_:
                GraphicsContext::CopyDataToBuffer(ChromaTrueDataBuffer.Get().get(), offset, sizeof(ChromaTrueBuffer),
&m_ChromaTrueBuffer);
                break;
```

```
            case CHROMABER_STYLE_:
                GraphicsContext::CopyDataToBuffer(ChromaStylizedDataBuffer.Get().get(), offset,
sizeof(ChromaStylizedBuffer), &m_ChromaStylizedBuffer);
                break;
            case COLOURGRAD_:
                GraphicsContext::CopyDataToBuffer(ColourGradDataBuffer.Get().get(), offset, sizeof(ColourGradBuffer),
&m_ColourGradBuffer);
                break;
            case GLASS_:
                GraphicsContext::CopyDataToBuffer(GlassDataBuffer.Get().get(), offset, sizeof(GlassBuffer),
&m_GlassBuffer);
                break;
            case EMBOS_:
                GraphicsContext::CopyDataToBuffer(EmbossDataBuffer.Get().get(), offset, sizeof(EmbossBuffer),
&m_EmbossBuffer);
                break;
            case BLUR_GAU_:
                GraphicsContext::CopyDataToBuffer(BlurGauDataBuffer.Get().get(), offset, sizeof(BlurGauBuffer),
&m_BlurGauBuffer);
                break;


            default:
                GraphicsContext::CopyDataToBuffer(ScreenDataBuffer.Get().get(), offset, sizeof(ScreenBuffer),
&m_ScreenBuffer);


                break;
            }


    }
```

EndCopyToBuffer() — Upload Active Post-Processing Buffer to GPU

Purpose

This function finalizes the frame setup by copying CPU-side post-processing effect data (e.g., scanline strength, vignette radius, etc.) into the currently active GPU buffer associated with the enabled effect. It is usually called at the end of frame preparation, just before rendering.

How It Works

Offset is set to zero: All data is copied to the beginning of each buffer.
```
int offset = 0;
```

The function then selects the active post-processing effect via the CURRENT_STATE_ switch:
```
switch (CURRENT_STATE_)
```

This is based on a shader state enum (ShaderToggles) such as PIXELATE_, BLOOM_, etc.
For the active effect, the corresponding CPU-side buffer (e.g., m_PixelBuffer, m_VignettingBuffer) is copied to its associated MultiResource GPU buffer via:
```
GraphicsContext::CopyDataToBuffer(buffer, offset, size, &source);
```

If none of the predefined post-processing states match, it defaults to uploading the ScreenBuffer.

supported Shader States and Corresponding Buffers
Shader Toggle, GPU Buffer, CPU Struct
PIXELATE_, pixelDataBuffer, m_PixelBuffer
POSTERIZE_, PosterizeDataBuffer, m_PosterizeBuffer
SCANLINE_, ScanlineDataBuffer, m_ScanlineBuffer
VIGNETTE_, VignettingDataBuffer, m_VignettingBuffer
BLOOM_, Bloom2DataBuffer, m_Bloom2Buffer
CHROMABER_TRUE_, ChromaTrueDataBuffer, m_ChromaTrueBuffer
CHROMABER_STYLE_, ChromaStylizedDataBuffer, m_ChromaStylizedBuffer
COLOURGRAD_, ColourGradDataBuffer, m_ColourGradBuffer
GLASS_, GlassDataBuffer, m_GlassBuffer
EMBOS_, EmbossDataBuffer, m_EmbossBuffer
BLUR_GAU_, BlurGauDataBuffer, m_BlurGauBuffer
(default), ScreenDataBuffer, m_ScreenBuffer

```
   void DrawVBIB_SetInlineResourceViewGraphics() {
        //m_ScanlineBuffer = ScanlineBuffer(LineThickness_, DimmedFactor_, transferPower_, vertical_);
```

```
            //RenderCommand::SetInlineResourceViewGraphics(4, PosterizeDataBuffer.Get().get(), posterize_desc,
ViewAccessType_ConstantBuffer);

            ////switch (EFFECT_PERMUTATIONS)
            switch (CURRENT_STATE_)
            {
            case PIXELATE_:
                RenderCommand::SetInlineResourceViewGraphics(4, pixelDataBuffer.Get().get(), pixel_desc,
ViewAccessType_ConstantBuffer);
                break;
            case POSTERIZE_:
                RenderCommand::SetInlineResourceViewGraphics(4, PosterizeDataBuffer.Get().get(), posterize_desc,
ViewAccessType_ConstantBuffer);
                break;
            case SCANLINE_:
                RenderCommand::SetInlineResourceViewGraphics(4, ScanlineDataBuffer.Get().get(), scanline_desc,
ViewAccessType_ConstantBuffer);
                break;
            case VIGNETTE_:
                RenderCommand::SetInlineResourceViewGraphics(4, VignettingDataBuffer.Get().get(), vignetting_desc,
ViewAccessType_ConstantBuffer);
                break;
            case BLOOM_:
                //RenderCommand::SetInlineResourceViewGraphics(4, BloomDataBuffer.Get().get(), bloom_desc,
ViewAccessType_ConstantBuffer);
                RenderCommand::SetInlineResourceViewGraphics(4, Bloom2DataBuffer.Get().get(), bloom2_desc,
ViewAccessType_ConstantBuffer);
                break;
            case CHROMABER_TRUE_:
                RenderCommand::SetInlineResourceViewGraphics(4, ChromaTrueDataBuffer.Get().get(), ChromaTrue_desc,
ViewAccessType_ConstantBuffer);
                break;
            case CHROMABER_STYLE_:
                RenderCommand::SetInlineResourceViewGraphics(4, ChromaStylizedDataBuffer.Get().get(),
ChromaStylized_desc, ViewAccessType_ConstantBuffer);
                break;
            case COLOURGRAD_:
                RenderCommand::SetInlineResourceViewGraphics(4, ColourGradDataBuffer.Get().get(), colourGrad_desc,
ViewAccessType_ConstantBuffer);
                break;
            case GLASS_:
                RenderCommand::SetInlineResourceViewGraphics(4, GlassDataBuffer.Get().get(), glass_desc,
ViewAccessType_ConstantBuffer);
                break;
            case EMBOS_:
                RenderCommand::SetInlineResourceViewGraphics(4, EmbossDataBuffer.Get().get(), Emboss_desc,
ViewAccessType_ConstantBuffer);
                break;
            case BLUR_GAU_:
                RenderCommand::SetInlineResourceViewGraphics(4, BlurGauDataBuffer.Get().get(), blurGau_desc,
ViewAccessType_ConstantBuffer);
                break;

            default:
                RenderCommand::SetInlineResourceViewGraphics(4, ScreenDataBuffer.Get().get(), screen_desc,
ViewAccessType_ConstantBuffer);

                break;
            }

    }
```

DrawVBIB_SetInlineResourceViewGraphics() — Bind Per-Effect GPU Buffer as Inline Constant Buffer

---

Purpose

This function selects and binds the appropriate GPU-side constant buffer for the currently enabled post-processing effect, using a shader slot (register) during the graphics pipeline draw call.
It acts as a resource view dispatcher, driven by the current shader state (CURRENT_STATE_).

```
-----------------------------------------------------------------------------------------------------------
Binding Mechanism

Each effect has a buffer and a buffer view (BufferViewDesc) prepared during initialization. This function uses:
RenderCommand::SetInlineResourceViewGraphics(
    slotIndex,           // e.g. 4
    bufferRef,           // GPU buffer (e.g. ScanlineDataBuffer)
    bufferViewDesc,      // Predefined view describing buffer layout
    ViewAccessType       // ViewAccessType_ConstantBuffer
);

This binds a GPU buffer as a constant buffer at graphics shader register slot 4.

-----------------------------------------------------------------------------------------------------------
Supported Effects and Bindings

Shader Toggle, GPU Buffer, View Descriptor
PIXELATE_, pixelDataBuffer, pixel_desc
POSTERIZE_, PosterizeDataBuffer, posterize_desc
SCANLINE_, ScanlineDataBuffer, scanline_desc
VIGNETTE_, VignettingDataBuffer, vignetting_desc
BLOOM_, Bloom2DataBuffer, bloom2_desc
CHROMABER_TRUE_, ChromaTrueDataBuffer, ChromaTrue_desc
CHROMABER_STYLE_, ChromaStylizedDataBuffer, ChromaStylized_desc
COLOURGRAD_, ColourGradDataBuffer, colourGrad_desc
GLASS_, GlassDataBuffer, glass_desc
EMBOS_, EmbossDataBuffer, Emboss_desc
BLUR_GAU_, BlurGauDataBuffer, blurGau_desc
(default fallback), ScreenDataBuffer, screen_desc
Each case ensures that the appropriate constant buffer view is bound, matching the currently active shader.


-----------------------------------------------------------------------------------------------------------
Notes

Uses slot 4 uniformly for all effects — make sure shaders expect the constant buffer at this slot.
Falls back to ScreenDataBuffer if no specific effect is active.
Commented out Bloom1 this was accidental - the shader works just ran out of time to put it in
```

Penny Values

```
Skateboard::ShaderInputLayoutRef RootSig;

//pipeline permutations
std::array<Skateboard::PipelineRef, EFFECT_PERMUTATIONS> Pipelines;

//normal rendering
Skateboard::PipelineRef NormalVisualizer;

 Skateboard::MultiResource<Skateboard::BufferRef> TriangleDataBuffer;
 Skateboard::MultiResource<Skateboard::BufferRef> InstanceDataBuffer;
 Skateboard::MultiResource<Skateboard::BufferRef> LightDataBuffer;
```

```
// post-processing buffers

size_t pennyBufferSize = 64 * 1024; //

//Buffer refs
Skateboard::MultiResource<Skateboard::BufferRef> ScreenDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> ScanlineDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> VignettingDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> PosterizeDataBuffer;

Skateboard::MultiResource<Skateboard::BufferRef> BloomDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> Bloom2DataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> BlurGauDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> ChromaTrueDataBuffer;

Skateboard::MultiResource<Skateboard::BufferRef> ChromaStylizedDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> ColourGradDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> EmbossDataBuffer;
Skateboard::MultiResource<Skateboard::BufferRef> GlassDataBuffer;

Skateboard::MultiResource<Skateboard::BufferRef> pixelDataBuffer;
```

```
// desc
BufferViewDesc screen_desc;
BufferViewDesc scanline_desc;
BufferViewDesc vignetting_desc;
BufferViewDesc posterize_desc;

BufferViewDesc bloom_desc;
BufferViewDesc bloom2_desc;
BufferViewDesc blurGau_desc;
BufferViewDesc ChromaTrue_desc;

BufferViewDesc ChromaStylized_desc;
BufferViewDesc colourGrad_desc;
BufferViewDesc Emboss_desc;
BufferViewDesc glass_desc;

BufferViewDesc pixel_desc;
```

```
// init buffer structs
ScreenBuffer m_ScreenBuffer = ScreenBuffer();
ScanlineBuffer m_ScanlineBuffer = ScanlineBuffer();
VignettingBuffer m_VignettingBuffer = VignettingBuffer();
PosterizeBuffer m_PosterizeBuffer = PosterizeBuffer();

BloomBuffer     m_BloomBuffer = BloomBuffer();
Bloom2Buffer    m_Bloom2Buffer = Bloom2Buffer();
BlurGauBuffer m_BlurGauBuffer = BlurGauBuffer();
ChromaTrueBuffer m_ChromaTrueBuffer = ChromaTrueBuffer();

ChromaStylizedBuffer m_ChromaStylizedBuffer = ChromaStylizedBuffer();
ColourGradBuffer m_ColourGradBuffer = ColourGradBuffer();
EmbossBuffer m_EmbossBuffer = EmbossBuffer();
GlassBuffer m_GlassBuffer = GlassBuffer();

PixelBuffer m_PixelBuffer = PixelBuffer();
```

```
// Pipeline default sate inti
ShaderToggles CURRENT_STATE_ = DEFAULT_STATE_;//
```

```
//--------------
//inline view desc
BufferViewDesc sbvdesc;
BufferViewDesc lightsbvdesc;
BufferViewDesc cbvdesc;
```

```
//resterizer
SamplerDesc StaticSamplerDesc = SamplerDesc::InitAsDefaultTextureSampler();
```

```
  //counts offsets for dynamic data uploaded every frame
  size_t m_Offset = ROUND_UP(sizeof(FrameData), GraphicsConstants::CONSTANT_BUFFER_ALIGNMENT);
```

```
//forked on each call with a unique data pointer/ otherwise default instance data is used
uint32_t m_InstanceDataForks = 0;
```

```
Frame m_Frame{ .AmbientLight = { 0.1, 0.1, 0.1} };
FrameData m_CameraData{};
```

```
std::vector<Light> m_Lights;
```

```
InstanceData m_DefaultInstanceData = InstanceData();
```

```
uint32_t m_DefaultTextureIDX = 0;
```

```
bool m_Pipeline_dirty;
bool m_VisualiseNormals = false;// probs delete
```

```
//effects - list stuff
std::vector < SavedEffect> savedEffect_List; // max4 for now
```

```
//Output Texture
MultiResource<TextureBufferRef> OutputTexture;
MultiResource<RenderTargetViewRef> OutputRTV;
MultiResource<TextureSRVRef>  OutputSRV;

MultiResource<TextureBufferRef> OutputTextures_0;
MultiResource<RenderTargetViewRef> OutputRTVs_0;
MultiResource<TextureSRVRef>  OutputSRVs_0;

MultiResource<TextureBufferRef> OutputTextures_[2];
MultiResource<RenderTargetViewRef> OutputRTVs_[2];
MultiResource<TextureSRVRef>  OutputSRVs_[2];
```

```
  //test values

  InstanceData current_instancedata;

  RasterizationPipelineDesc Raster{};
```

```
int previewscreenSize = 2;
```

```
float aberrationFactor_XY = 0.5;
bool link_XY = true;
```

`float colC[4] = {0.8,0.5,0.9,1};`	
`float chromaticWeights1[4] = {0.8,0.5,0,1};` `float chromaticWeights2[4] = {0.8,0.5,0,1};`	
`float startColor_CG[3] = { 1,1,1};` `float endColor_CG[3] = { 1.2, 0.01, 0.51 };`	

`float4 chromaticOffset1;`
`float4 chromaticOffset2;`
`float4 chromaticOffset_XY;`
`bool offsetlinked = true;`
`bool XYlinked = true;`	

`bool panelinked = true;` `bool pixelinked = true;`	
`float weight_all;` `float multi_all;`	
`bool link_weight;` `bool link_Multiplier;`	

`float4 SpecularC = float4(1, 1, 1, 1);`
`float4 DiffuseC = float4(1, 1, 1, 1);`
`float SpecularWeight = 1;`
`float SpecularPower = 200;`	

`std::vector<pen::Vertex> vertices_screen =`
`{`
`{ float3(-0.5,-0.75,0) ,float3(1,0,0) ,float2(0,1) - float2(-0.05,0.05) , float3(0,0,-1)},`
`{ float3(0.5,-0.75,0) ,float3(0,2,0) ,float2(1,1) - float2(-0.05,0.05) , float3(0,0,-1)},`
`{ float3(0,0.5,0) ,float3(0,0,3) ,float2(0.5, 0) - float2(-0.05,0.05), float3(0,0,-1)}`
`};`	

The Renderer

```
//renderer + pipeline stuff

void PennyRender::Init() {

    SKTBD_APP_INFO("Renderer test 2 Init")

    //Layout
     ShaderInputLayoutDesc Layout{};


     //instance index //0- slot
     Layout.AddRootConstant(1, 0);


     //frame data // 1- slot
     Layout.AddConstantBufferView(1);


     //instance data // 2- slot
     Layout.AddShaderResourceView(0);


     //light data // 3- slot
     Layout.AddShaderResourceView(1);
```

```
    //--

    //all post-processing buffers data // 4 - slot
    Layout.AddConstantBufferView(2);


    Layout.AddStaticSampler(StaticSamplerDesc, 0);


    Layout.DescriptorsDirctlyAddresssed = true;
    Layout.SamplersDirectlyAddressed = true;
    Layout.CanUseInputAssembler = true;


    Init_RasterPipeline(Layout);//


    renderToScreen_init();//// initilising render to target
    renderToScreen_init_pass(OutputTextures_0, OutputRTVs_0, OutputSRVs_0);// - not in use -- suppost to be for multi-
render passes

}
```

PennyRender::Init() — Pipeline and Renderer Initialization

Purpose

This function initializes the shader input layout, post-processing pipeline, and render targets for the rendering system used by the PennyRender renderer. It sets up all required resource bindings, including constant buffers, samplers, and resource views, and then calls initialization routines for the graphics raster pipeline and render-to-screen pipeline.

Function Breakdown

Logging
SKTBD_APP_INFO("Renderer test 2 Init")

Logs the initialization event for debugging or performance tracking.

Shader Input Layout Setup

ShaderInputLayoutDesc Layout{};

A layout descriptor is created to define how shaders will receive data (CBVs, SRVs, root constants, etc.).

Resource Slot Bindings:
Shader Slot, Type, Resource Use
0, RootConstant, Instance index
1, ConstantBufferView, Per-frame data (e.g., camera)
2, ShaderResourceView, Instance data buffer
3, ShaderResourceView, Light data buffer
4, ConstantBufferView, Post-processing data buffer
0 (Sampler), StaticSampler, Texture sampling

```
Layout.AddRootConstant(1, 0);
Layout.AddConstantBufferView(1);  // Frame data (e.g., camera)
Layout.AddShaderResourceView(0);  // Instance data (e.g., transform matrices)
Layout.AddShaderResourceView(1);  // Light data
Layout.AddConstantBufferView(2);  // Post-processing buffer (e.g., bloom, vignette)
Layout.AddStaticSampler(StaticSamplerDesc, 0);
```

Shader Binding Model Configuration

```
Layout.DescriptorsDirctlyAddresssed = true;
Layout.SamplersDirectlyAddressed = true;
Layout.CanUseInputAssembler = true;
```

These flags determine how shaders access their resources:
Direct addressing of descriptors (not through descriptor tables).
Input assembler is enabled, allowing use of vertex/index buffers.

Pipeline Initialization

Init_RasterPipeline(Layout);

Initializes the rasterization pipeline with the defined shader layout. This configures the graphics pipeline (shaders, buffers, states) to prepare for rendering.

Render Target Initialization

renderToScreen_init(); // Initializes base render-to-texture resources

renderToScreen_init_pass(OutputTextures_0, OutputRTVs_0, OutputSRVs_0);

Sets up render targets:
OutputTextures_0: The textures to render into.
OutputRTVs_0: Render Target Views.
OutputSRVs_0: Shader Resource Views for sampling the output in post-processing.
Note: The second call is currently marked as "not in use" — originally intended for multi-pass post-processing setups.

Conceptual Summary

This function configures the graphics pipeline to accept structured and consistent input from CPU-side buffers and sets up output targets for screen rendering. It's a crucial part of the initialization flow that ensures shaders receive the right data in the right slots and that the rendering output can be displayed or post-processed.

Key Features Enabled

Dynamic instance rendering (via root constants + instance SRVs).
Per-frame camera and lighting data.
Slot for post-processing constant buffers (e.g., bloom, vignette).
Support for static samplers.
Input assembler support (for vertex/indexed geometry).
Configurable render targets.

```
void PennyRender::Init_RasterPipeline(ShaderInputLayoutDesc Layout) {

    RootSig = ResourceFactory::CreateShaderInputLayout(Layout);

    //RasterizationPipelineDesc Raster{};
    //Raster{};
    Raster.Rasterizer = RasterizerConfig::Default();
    Raster.Blend.AlphaToCoverage = false;
    Raster.Blend.IndependentBlendEnable = false;
    Raster.Blend.RTBlendConfigs[0].BlendEnable = false;
    Raster.Blend.RTBlendConfigs[0].RenderTargetWriteMask = 0xF;
    Raster.DepthStencil.DepthEnable = false;
    Raster.DepthStencil.BackFace.StencilDepthFailOp = SKTBD_StencilOp_KEEP;
    Raster.DepthStencil.BackFace.StencilFailOp = SKTBD_StencilOp_KEEP;
    Raster.DepthStencil.BackFace.StencilFunc = SKTBD_CompareOp_ALWAYS;
    Raster.DepthStencil.BackFace.StencilPassOp = SKTBD_StencilOp_KEEP;
    Raster.DepthStencil.DepthFunc = SKTBD_CompareOp_LESS;
    Raster.DepthStencil.DepthWriteAll = true;
    Raster.DepthStencil.FrontFace = Raster.DepthStencil.BackFace;
    Raster.DepthStencil.StencilEnable = false;
    Raster.DepthstencilTargetFormat = DataFormat_DEFAULT_DEPTHSTENCIL;
    Raster.InputPrimitiveType = PrimitiveTopologyType_Triangle;
    Raster.RenderTargetCount = 1;
    Raster.RenderTargetDataFormats[0] = DataFormat_DEFAULT_BACKBUFFER;
    Raster.InputVertexLayout = Vertex::VertexLayout();
    Raster.SampleCount = 1;
    Raster.SampleQuality = 0;
    Raster.SampleMask = 1;

    //initilising all shaders
    Raster.SetVertexShader(L"CMP203_VertexShader");
    Raster.SetPixelShader(L"CMP203_PixelShader_Unlit");
    //Raster.SetPixelShader(L"test_ps");
```

```
    //piepline permutations
    Init_ShaderPermutation(Raster);


    //NormalPipeline


    Raster.DepthStencil.DepthEnable = true;
    Raster.SetGeometryShader(L"CMP203_NormalGS_GS");
    Raster.SetVertexShader(L"CMP203_NormalGS_VS");
    Raster.SetPixelShader(L"CMP203_NormalGS_PS");



    //creating PipelineDesc -
    PipelineDesc PiplDesc;
    PiplDesc.GlobalLayoutSignature = RootSig;
    PiplDesc.Type = PipelineType_Graphics;
    PiplDesc.TypeDesc = &Raster;


    NormalVisualizer = ResourceFactory::CreatePipelineState(PiplDesc);


    //create Buffers
    //buffer desc
    BufferDesc bufferdesc{};


    //Triangle Buffer
    bufferdesc.Init(DynamicBufferSize, ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead);
    TriangleDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer(bufferdesc); });


    //InstanceData Buffer
    bufferdesc.Init(InstanceDataBufferSize, ResourceAccessFlag_CpuWrite | ResourceAccessFlag_GpuRead);
    InstanceDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer(bufferdesc); });


    //LightDataBuffer
    LightDataBuffer.ForEach([&](BufferRef& ref) {ref = ResourceFactory::CreateBuffer(bufferdesc); });


    //-- postprocessing buffers
    InitBuffers(bufferdesc);


    //Create Views
    //Camera data
    cbvdesc.InitAsConstantBuffer<Frame>(0);


    //InstanceData
    sbvdesc.InitAsStructuredBuffer<InstanceData>(0, InstanceDataBufferSize / sizeof(InstanceData));


    //lightdata
    lightsbvdesc.InitAsStructuredBuffer<Light>(0, InstanceDataBufferSize / sizeof(Light));


    //-- postprocessing data desc
    InitData(InstanceDataBufferSize);


    //compute default camera
    auto aspect = GraphicsContext::GetClientAspectRatio();
    m_CameraData.ProjectionMatrix = glm::perspectiveLH(glm::radians(90.f), aspect, 0.01f, 100.f);
    m_CameraData.ViewMatrix = glm::lookAtLH(glm::vec3(0, 0, -10), glm::vec3(0, 0, 0), glm::vec3(0, 1, 0));


    m_DefaultInstanceData.TextureIndex = m_DefaultTextureIDX;


    //default Instance data being past to buffer
    GraphicsContext::CopyDataToBuffer(InstanceDataBuffer[0].get(), 0, sizeof(InstanceData), &m_DefaultInstanceData);
    GraphicsContext::CopyDataToBuffer(InstanceDataBuffer[1].get(), 0, sizeof(InstanceData), &m_DefaultInstanceData);
    GraphicsContext::CopyDataToBuffer(InstanceDataBuffer[2].get(), 0, sizeof(InstanceData), &m_DefaultInstanceData);

}
```

```
PennyRender::Init_RasterPipeline(ShaderInputLayoutDesc Layout)
```

Purpose

This function configures and initializes the graphics pipeline, including the shader input layout, rasterizer settings, render target formats, and key GPU buffers required for rendering and post-processing effects. It also sets up a dedicated normal visualization pipeline and initializes associated structured and constant buffers.

Function Breakdown

---

1. Root Signature Creation

```
RootSig = ResourceFactory::CreateShaderInputLayout(Layout);
```

Creates the shader root signature from the layout passed by PennyRender::Init() — defining how data is bound to the shaders.

---

2. Rasterization Pipeline Configuration

```
Raster.Rasterizer = RasterizerConfig::Default();
...
Raster.InputVertexLayout = Vertex::VertexLayout();
```

Sets up the rasterization pipeline with the following configurations:
No blending, no depth stencil, default backbuffer format.
Triangle-based topology.
Uses a default vertex layout for geometry rendering.
Render output is single-sample (no MSAA).
Shaders assigned:
Vertex Shader: CMP203_VertexShader
Pixel Shader: CMP203_PixelShader_Unlit

```
Raster.SetVertexShader(...);
Raster.SetPixelShader(...);
```

---

3. Pipeline Shader Permutations

```
Init_ShaderPermutation(Raster);
```

Handles setup for various shader configurations (e.g., toggling between effects or shader versions).

---

4. Normal Visualization Pipeline

```
Raster.DepthStencil.DepthEnable = true;
Raster.SetGeometryShader(L"CMP203_NormalGS_GS");
Raster.SetVertexShader(L"CMP203_NormalGS_VS");
Raster.SetPixelShader(L"CMP203_NormalGS_PS");
```

This alternate configuration enables depth testing and sets geometry shaders for visualizing vertex normals in 3D.

---

5. Pipeline State Creation

```
PipelineDesc PiplDesc;
...
NormalVisualizer = ResourceFactory::CreatePipelineState(PiplDesc);
```

Creates the finalized pipeline state object (PSO) with the configured raster pipeline and root signature. This is stored in NormalVisualizer.

---

6. Buffer Initialization

Initializes GPU-side structured buffers for rendering geometry and lights:
TriangleDataBuffer
InstanceDataBuffer
LightDataBuffer

```
bufferdesc.Init(...);
TriangleDataBuffer.ForEach(...);

All buffers are marked as CpuWrite | GpuRead for frequent updates from the CPU side.

---------------------------------------------------------------------
7. Post-Processing Buffers
---------------------------------------------------------------------

InitBuffers(bufferdesc);

Initializes all post-processing effect buffers (e.g., vignette, posterize, bloom), using the same access pattern as
above.

---------------------------------------------------------------------
8. Buffer View Descriptors
---------------------------------------------------------------------

Sets up views for constant/structured buffers, describing how GPU shaders access each buffer:
cbvdesc.InitAsConstantBuffer<Frame>(0);
sbvdesc.InitAsStructuredBuffer<InstanceData>(...);
lightsbvdesc.InitAsStructuredBuffer<Light>(...);
InitData(...); // Additional post-process buffer view init

Each descriptor (CBV/SBV) defines:
The buffer type.
The offset and size (based on buffer capacity).
How it's accessed by the GPU.

---------------------------------------------------------------------
9. Default Camera Setup
---------------------------------------------------------------------

m_CameraData.ProjectionMatrix = glm::perspectiveLH(...);
m_CameraData.ViewMatrix = glm::lookAtLH(...);

Creates a default view-projection matrix:
Perspective projection with 90° FOV.
Camera located at (0, 0, -10) looking at origin.

---------------------------------------------------------------------
10. Default Instance Data
---------------------------------------------------------------------

m_DefaultInstanceData.TextureIndex = m_DefaultTextureIDX;

Initializes a fallback InstanceData structure, assigning the default texture index for all instances.
GraphicsContext::CopyDataToBuffer(...);

Copies default instance data into three GPU buffers — triple-buffered for use across frames.

---------------------------------------------------------------------
Conceptual Summary

Init_RasterPipeline fully configures the rendering pipeline:
Establishes GPU-side shader data bindings.
Builds pipelines for both base rendering and normal visualization.
Allocates and binds buffers for all geometry, lighting, and post-processing data.
Sets up initial camera and rendering state.
It is one of the core setup functions responsible for getting the renderer ready to draw frames efficiently and flexibly.
```

```
void PennyRender::Init_ShaderPermutation(RasterizationPipelineDesc Raster) {


    //piepline permutations


    for (uint16_t p = 0; p < EFFECT_PERMUTATIONS; p++)
    {
        auto RSPD = Raster;
        RSPD.DepthStencil.DepthEnable = true;


        ////pipeling permutaions
```

```
        //switch (CURRENT_STATE_Pass[p])
        switch ((ShaderToggles)p)
        {
        case PIXELATE_:
            RSPD.SetPixelShader(L"pixelate_ps");
            break;
        case POSTERIZE_:
            RSPD.SetPixelShader(L"posterize_ps");
            break;
        case SCANLINE_:
            RSPD.SetPixelShader(L"scanline_ps");
            break;
        case VIGNETTE_:
            RSPD.SetPixelShader(L"vignetting_ps");
            break;
        case BLOOM_:
            //RSPD.SetPixelShader(L"bloom_ps");
            RSPD.SetPixelShader(L"bloom2_ps");// needs to be passed through the blur shader
            break;
        case CHROMABER_TRUE_:
            RSPD.SetPixelShader(L"ChromaAber_True_ps");
            break;
        case CHROMABER_STYLE_:
            RSPD.SetPixelShader(L"ChromaAber_Stylized_ps");
            break;
        case COLOURGRAD_:
            RSPD.SetPixelShader(L"colourGrading_ps");
            break;
        case GLASS_:
            RSPD.SetPixelShader(L"glass_ps");
            break;
        case EMBOS_:
            RSPD.SetPixelShader(L"embossed_ps");
            break;
        case BLUR_GAU_:
            RSPD.SetPixelShader(L"blur_gaussian_ps");
            break;
        case POSTPROSSESSING_:
            RSPD.SetPixelShader(L"Defualt_ps");
            break;
        default:
            RSPD.SetPixelShader(L"Defualt_ps");
            //RSPD.SetPixelShader(L"CMP203_NormalGS_PS");
            break;
        }

        PipelineDesc PiplDesc;
        PiplDesc.GlobalLayoutSignature = RootSig;
        PiplDesc.Type = PipelineType_Graphics;
        PiplDesc.TypeDesc = &RSPD;
        Pipelines[p] = ResourceFactory::CreatePipelineState(PiplDesc);
        //Pipelines[0] = ResourceFactory::CreatePipelineState(PiplDesc);

    }

}
```

PennyRender::Init_ShaderPermutation(RasterizationPipelineDesc Raster)

Purpose

Initializes and stores a set of graphics pipeline permutations, each using a different pixel shader corresponding to a specific post-processing effect. This system allows fast switching between visual effects at runtime by pre-building the necessary pipeline state objects (PSOs).

🦴 Function Overview

Loop Through Effect Permutations

for (uint16_t p = 0; p < EFFECT_PERMUTATIONS; p++)

Iterates over each defined effect in EFFECT_PERMUTATIONS — a constant that represents the number of supported post-processing effects or visual shader modes.

Base Raster Configuration

auto RSPD = Raster;
RSPD.DepthStencil.DepthEnable = true;

Creates a copy of the input raster pipeline configuration. Then, enables depth testing for each effect permutation (useful for z-ordering or 3D-aware effects).

Set Pixel Shader Based on Effect

A switch is used to assign the correct pixel shader for each permutation index p, interpreted as a ShaderToggles enum.
Example mappings:
Shader Toggle Enum, Assigned Pixel Shader
PIXELATE_, pixelate_ps
POSTERIZE_, posterize_ps
SCANLINE_, scanline_ps
BLOOM_, bloom2_ps
CHROMABER_TRUE_, ChromaAber_True_ps
CHROMABER_STYLE_, ChromaAber_Stylized_ps
COLOURGRAD_, colourGrading_ps
GLASS_, glass_ps
EMBOS_, embossed_ps
BLUR_GAU_, blur_gaussian_ps
POSTPROSSESSING_/other, Defualt_ps
This allows each visual effect to have its dedicated rendering pipeline with appropriate shading logic.

Create Pipeline State for Each Shader

PipelineDesc PiplDesc;
PiplDesc.GlobalLayoutSignature = RootSig;
PiplDesc.Type = PipelineType_Graphics;
PiplDesc.TypeDesc = &RSPD;
Pipelines[p] = ResourceFactory::CreatePipelineState(PiplDesc);

For each shader permutation:
Constructs a PipelineDesc structure referencing the layout (RootSig) and configured raster data (RSPD).
Calls the GPU abstraction layer to create a pipeline state object (PSO).
Stores the resulting PSO in the Pipelines array indexed by effect type.

Output

The result is a populated Pipelines[] array with prebuilt PSOs, each optimized for a specific post-processing effect. This avoids runtime compilation and enables rapid switching between visual modes during rendering.

Conceptual Summary

This function is a core part of the effect system, handling:
Initialization of shader permutations.
Binding of pixel shaders specific to visual styles.
Creation of PSO objects for all valid rendering paths.
This enables PennyRender to dynamically switch effects via CURRENT_STATE_ without incurring pipeline creation cost during frame rendering.

```
void PennyRender::Begin()
{
    //Move Onto Next frame In Buffer
    TriangleDataBuffer.IncrementCounter();


    InstanceDataBuffer.IncrementCounter();
```

```
    LightDataBuffer.IncrementCounter();


    BeginIncrementCounter();//


    //reset the buffer Offset
    m_Offset = ROUND_UP(sizeof(Frame), GraphicsConstants::CONSTANT_BUFFER_ALIGNMENT);



    m_InstanceDataForks = 1;


    RenderCommand::SetInputLayoutGraphics(RootSig.get());
    //RenderCommand::SetPipelineState(Pipelines[0].get());
    //RenderCommand::SetPipelineState(Pipelines[m_PipelineSelection].get());
    RenderCommand::SetPipelineState(Pipelines[CURRENT_STATE_].get());
}
```

PennyRender::Begin()

Purpose

The Begin() function prepares the renderer for the next frame by:
Incrementing counters to move to the next frame.
Setting up the necessary buffers and resources.
Configuring the graphics pipeline state for the current frame.

Function Overview

Incrementing Data Buffers

```
TriangleDataBuffer.IncrementCounter();
InstanceDataBuffer.IncrementCounter();
LightDataBuffer.IncrementCounter();
BeginIncrementCounter();
```

TriangleDataBuffer, InstanceDataBuffer, and LightDataBuffer are incremented to move to the next frame in their respective buffers. This ensures the renderer works with the latest data during rendering.
BeginIncrementCounter() is called to handle additional internal counters (for other buffers related to rendering).

Resetting the Buffer Offset

```
m_Offset = ROUND_UP(sizeof(Frame), GraphicsConstants::CONSTANT_BUFFER_ALIGNMENT);
```

The offset is reset to ensure that the frame data is correctly aligned to the GPU's constant buffer alignment requirements. This ensures that when new data is copied into the buffer, it aligns with the expected memory boundaries.

Setting the Instance Data Forks

```
m_InstanceDataForks = 1;
```

m_InstanceDataForks is set to 1. This manages how many copies (or "forks") of the instance data will be used, typically one for each object in a scene.

Setting the Graphics Pipeline Input Layout

```
RenderCommand::SetInputLayoutGraphics(RootSig.get());
```

Input layout is configured using the Root Signature (RootSig), which is a descriptor of the pipeline’s input resources (constant buffers, resources, etc.).
This function ensures that the GPU knows the layout of incoming vertex data.

Setting the Graphics Pipeline State

```
RenderCommand::SetPipelineState(Pipelines[CURRENT_STATE_].get());
```

The pipeline state for the current frame is set using the Pipelines[CURRENT_STATE_].

The CURRENT_STATE_ is an enum variable that selects the current rendering effect, enabling the appropriate shader/pipeline configuration.

Output
The function does not return anything but prepares the renderer for the upcoming frame by:
Ensuring buffers are properly incremented and prepared.
Setting the input layout and pipeline state to match the current rendering configuration.

Conceptual Summary

The Begin() function is responsible for preparing the renderer for the next frame, ensuring that:
Data buffers are updated for the next frame.
The GPU has the necessary information on how to interpret vertex and constant data.
The correct pipeline state is set to handle the current rendering state, as defined by CURRENT_STATE_.

```
void PennyRender::End()
{
    m_Frame.Matrices = m_CameraData;
    m_Frame.LightCount = m_Lights.size();
    m_Frame.CameraMatrix = glm::inverse(m_CameraData.ViewMatrix);


    //update the frame data buffer section
    GraphicsContext::CopyDataToBuffer(TriangleDataBuffer.Get().get(), 0, sizeof(Frame), &m_Frame);


    //send lightdata to gpu
    GraphicsContext::CopyDataToBuffer(LightDataBuffer.Get().get(), 0, sizeof(Light) * m_Frame.LightCount,
m_Lights.data());


    EndCopyToBuffer();


}
```

PennyRender::End()

Purpose

The End() function finalizes the frame rendering process by:
Updating the frame data with the camera and lighting information.
Copying updated frame data and light data to the GPU.
Completing any necessary buffer copy operations for the current frame.

Function Overview

Updating Frame Data

```
m_Frame.Matrices = m_CameraData;
m_Frame.LightCount = m_Lights.size();
m_Frame.CameraMatrix = glm::inverse(m_CameraData.ViewMatrix);
```

Camera and Lighting Data:

m_Frame.Matrices is updated with the camera data (m_CameraData), ensuring the frame has the latest view and projection matrix information.
m_Frame.LightCount is updated to reflect the number of lights in the scene (m_Lights.size()), allowing the shader to know how many lights to process.
m_Frame.CameraMatrix stores the inverse view matrix (glm::inverse(m_CameraData.ViewMatrix)), which will be used in shaders to transform world space coordinates into camera space.

Copying Updated Frame Data to GPU

```
GraphicsContext::CopyDataToBuffer(TriangleDataBuffer.Get().get(), 0, sizeof(Frame), &m_Frame);
```

The updated frame data (m_Frame) is copied to the GPU using the TriangleDataBuffer. This will allow the GPU to access the latest camera and lighting data during rendering.

The data is copied starting from offset 0, and the size of the data being copied is the size of the Frame structure.

Sending Light Data to GPU

```
GraphicsContext::CopyDataToBuffer(LightDataBuffer.Get().get(), 0, sizeof(Light) * m_Frame.LightCount, m_Lights.data());
```

Light data is copied to the GPU using the LightDataBuffer. The light data is passed to the shader, and the number of lights (m_Frame.LightCount) determines how much light data will be copied.
The light data is represented as an array (m_Lights.data()), and the size of the data copied is sizeof(Light) * m_Frame.LightCount.

Finalizing Buffer Copy Operations

```
EndCopyToBuffer();
```

EndCopyToBuffer() is called to perform any additional copying of buffer data necessary at the end of the frame.
This function handles any post-processing or final buffer updates required for the GPU.

Output

The function does not return anything but ensures that:
The frame data is updated and available for the next stage in the graphics pipeline.
The light data is sent to the GPU and will be used in shaders for lighting calculations.
The final buffer copy operations are completed to ensure the GPU has all necessary data.

Conceptual Summary
The End() function finalizes the rendering process by:
Updating the frame with the latest camera and light information.
Copying updated frame data and light data to GPU buffers.
Finalizing buffer copy operations to ensure the GPU is prepared for the next frame.

```
void PennyRender::renderToScreen_end(MultiResource<RenderTargetViewRef> OutputRTV, MultiResource<TextureSRVRef>  OutputSRV) {
////next frame
++OutputSRV;
++OutputRTV;
}
```

PennyRender::renderToScreen_end()

Purpose

The renderToScreen_end() function is used to finalize the process of rendering to the screen by updating the render target and shader resource view (SRV) for the next frame.

Function Overview

Incrementing the Render Target View (RTV)

```
++OutputRTV;
```

Purpose: This operation increments the Render Target View (RTV) to point to the next render target in the sequence.
Effect: This essentially prepares the system to render the next frame to a new target, moving forward in the sequence of render targets.

Incrementing the Shader Resource View (SRV)

```
++OutputSRV;
```

Purpose: Similar to the RTV, this operation increments the Shader Resource View (SRV) to the next texture in the sequence.
Effect: It ensures that the next texture resource (for example, a texture containing the result of the current frame's rendering) is used in subsequent shader operations, typically for post-processing or any other shader-based effects.

Conceptual Summary

The renderToScreen_end() function serves as a mechanism to increment the render target and shader resource view pointers, effectively preparing them for the next frame. This allows the system to move from one frame's output to the next frame's input seamlessly, ensuring continuous rendering.

```
void PennyRender::SetFrameData(FrameData newData)
{
m_CameraData = newData;
}
```

PennyRender::SetFrameData()

Purpose

The SetFrameData() function is used to update the camera data for the current frame. It takes a FrameData object as input and assigns it to the m_CameraData member variable.

Function Overview

Updating the Camera Data

m_CameraData = newData;

Purpose: This line of code assigns the provided newData (of type FrameData) to the m_CameraData member variable.
Effect: The camera data for the current frame is updated. This includes information such as the view and projection matrices, which are typically used for rendering the scene from the camera's perspective.

The primary effect is the updating of the m_CameraData variable, which will be used later during the rendering pipeline for camera-related calculations.

Conceptual Summary

The SetFrameData() function is a simple setter that allows external code to update the camera data for the current frame. This is important for scenarios where the camera might change during the course of the program, and new camera data needs to be used for rendering.

```
void PennyRender::DrawVertices(Vertex* vertexBuffer, size_t vertexcount, uint32_t* indexbuffer, size_t indexcount,
InstanceData* data, int CurrentRenderPass_in)
{
    size_t VBSize = vertexcount * Vertex::VertexLayout().GetStride();
    size_t IBSize = sizeof(int32_t) * indexcount;

    GraphicsContext::CopyDataToBuffer(TriangleDataBuffer.Get().get(), m_Offset, VBSize, vertexBuffer);
    GraphicsContext::CopyDataToBuffer(TriangleDataBuffer.Get().get(), m_Offset + VBSize, IBSize, indexbuffer);

    Skateboard::VertexBufferView vbv{};
    Skateboard::IndexBufferView ibv{};

    vbv.m_Offset = m_Offset;
    vbv.m_ParentResource = TriangleDataBuffer.Get();
    vbv.m_VertexCount = vertexcount;
    vbv.m_VertexStride = Vertex::VertexLayout().GetStride();

    ibv.m_Offset = m_Offset + VBSize;
    ibv.m_Format = IndexFormat::bit32;
    ibv.m_ParentResource = TriangleDataBuffer.Get();
    ibv.m_IndexCount = indexcount;

    m_Offset += VBSize + IBSize;

    DrawVBIB(&vbv, &ibv, data, CurrentRenderPass_in);
}
```

PennyRender::DrawVertices()

Purpose

The DrawVertices() function handles the process of copying vertex and index data to a buffer and then initiating the drawing process. It takes a vertex buffer, index buffer, instance data, and the current render pass as inputs. This function is primarily used to render geometry based on the provided data.

🦴 Function Overview

Calculating Buffer Sizes

```
size_t VBSize = vertexcount * Vertex::VertexLayout().GetStride();
size_t IBSize = sizeof(int32_t) * indexcount;
```

Purpose:

The size of the vertex buffer (VBSize) is calculated by multiplying the number of vertices (vertexcount) by the stride of each vertex (which can be obtained via Vertex::VertexLayout().GetStride()).
The size of the index buffer (IBSize) is calculated by multiplying the number of indices (indexcount) by the size of an index (assumed to be sizeof(int32_t)).

Copying Data to Triangle Data Buffer

```
GraphicsContext::CopyDataToBuffer(TriangleDataBuffer.Get().get(), m_Offset, VBSize, vertexBuffer);
GraphicsContext::CopyDataToBuffer(TriangleDataBuffer.Get().get(), m_Offset + VBSize, IBSize, indexbuffer);
```

Purpose:
Copies the vertex data to the TriangleDataBuffer starting from the offset m_Offset.
Then, copies the index data to the buffer at the new offset, which is m_Offset + VBSize.
Effect: The vertex and index data are placed into a GPU buffer for rendering.

Setting Up Vertex and Index Buffer Views

```
Skateboard::VertexBufferView vbv{};
Skateboard::IndexBufferView ibv{};
```

Purpose:
Initializes VertexBufferView (vbv) and IndexBufferView (ibv), which describe how to access the vertex and index data in the buffer.

Configuring Vertex Buffer View

```
vbv.m_Offset = m_Offset;
vbv.m_ParentResource = TriangleDataBuffer.Get();
vbv.m_VertexCount = vertexcount;
vbv.m_VertexStride = Vertex::VertexLayout().GetStride();
```

Purpose:
The VertexBufferView (vbv) is configured with the following:
m_Offset: The offset from where the vertex data starts in the buffer.
m_ParentResource: The buffer resource that holds the vertex data.
m_VertexCount: The total number of vertices.
m_VertexStride: The stride (size) of each vertex.

Configuring Index Buffer View

```
ibv.m_Offset = m_Offset + VBSize;
ibv.m_Format = IndexFormat::bit32;
ibv.m_ParentResource = TriangleDataBuffer.Get();
ibv.m_IndexCount = indexcount;
```

Purpose:
The IndexBufferView (ibv) is configured with the following:
m_Offset: The offset from where the index data starts in the buffer (m_Offset + VBSize).
m_Format: The format of the indices (bit32 indicates 32-bit indices).
m_ParentResource: The buffer resource that holds the index data.
m_IndexCount: The total number of indices.

Updating the Offset

```
m_Offset += VBSize + IBSize;
```

Purpose: The offset (m_Offset) is updated by adding the sizes of the vertex and index buffers. This ensures that the next time vertex or index data is copied, it will be placed in the correct location in the buffer.

Calling Draw Function

DrawVBIB(&vbv, &ibv, data, CurrentRenderPass_in);

Purpose: The DrawVBIB() function is called to initiate the drawing process with the configured vertex and index buffer views (vbv and ibv), the instance data (data), and the current render pass (CurrentRenderPass_in).

Output

Effect: This function initiates the drawing process by sending the vertex and index data to the GPU, using the TriangleDataBuffer. It sets up the buffers and calls the DrawVBIB() function to actually issue the draw command.
No return value.

Conceptual Summary

The DrawVertices() function is responsible for copying vertex and index data to a buffer in the GPU, configuring the necessary buffer views, updating the offset, and then triggering the drawing process. This is a typical function used in rendering pipelines to handle geometry rendering for a frame, with support for vertex and index data, as well as handling multiple render passes and instance data.

```
void PennyRender::DrawVBIB(VertexBufferView* vb, IndexBufferView* ib, InstanceData* data, int CurrentRenderPass_in)
{
    if (m_Pipeline_dirty)
    {
        m_Pipeline_dirty = false;
        //RenderCommand::SetPipelineState(Pipelines[0].get());
        //RenderCommand::SetPipelineState(Pipelines[m_PipelineSelection].get());
        RenderCommand::SetPipelineState(Pipelines[CURRENT_STATE_].get());
    }



    if (data)
    {
        GraphicsContext::CopyDataToBuffer(InstanceDataBuffer.Get().get(), sizeof(InstanceData) * m_InstanceDataForks,
sizeof(InstanceData), data);
        RenderCommand::SetInline32bitDataGraphics(0, &m_InstanceDataForks, 1);
        ++m_InstanceDataForks;
    }
    else
    {
        //default value
        int d = 0;
        RenderCommand::SetInline32bitDataGraphics(0, &d, 1);
    }


    RenderCommand::SetInlineResourceViewGraphics(1, TriangleDataBuffer.Get().get(), cbvdesc,
ViewAccessType_ConstantBuffer);
    RenderCommand::SetInlineResourceViewGraphics(2, InstanceDataBuffer.Get().get(), sbvdesc, ViewAccessType_GpuRead);
    RenderCommand::SetInlineResourceViewGraphics(3, LightDataBuffer.Get().get(), lightsbvdesc, ViewAccessType_GpuRead);


    DrawVBIB_SetInlineResourceViewGraphics();//


    RenderCommand::SetIndexBuffer(ib);
    RenderCommand::SetVertexBuffer(vb, 1);


    RenderCommand::SetPrimitiveTopology(Topology);


    RenderCommand::DrawIndexed(0, 0, ib->m_IndexCount);


    if (m_VisualiseNormals)//
    {
        RenderCommand::SetPipelineState(NormalVisualizer.get());
        RenderCommand::DrawIndexed(0, 0, ib->m_IndexCount);
        RenderCommand::SetPipelineState(Pipelines[CURRENT_STATE_].get());
    }



}
```

PennyRender::DrawVBIB()

Purpose

The DrawVBIB() function is responsible for managing the drawing process for a given set of vertex and index buffers. It ensures that pipeline states, instance data, and resources are properly set before issuing a draw call. This function handles both regular and instance-based rendering, allowing for dynamic and efficient drawing commands to be issued to the GPU.

Function Overview

Pipeline State Check

```
if (m_Pipeline_dirty)
{
    m_Pipeline_dirty = false;
    RenderCommand::SetPipelineState(Pipelines[CURRENT_STATE_].get());
}
```

The function first checks if the pipeline state is dirty (i.e., needs updating). If so, it sets the appropriate pipeline state using the current state from Pipelines[CURRENT_STATE_]. This ensures that any changes in the pipeline configuration are reflected before drawing.

Handling Instance Data

```
if (data)
{
    GraphicsContext::CopyDataToBuffer(InstanceDataBuffer.Get().get(), sizeof(InstanceData) * m_InstanceDataForks,
sizeof(InstanceData), data);
    RenderCommand::SetInline32bitDataGraphics(0, &m_InstanceDataForks, 1);
    ++m_InstanceDataForks;
}
else
{
    int d = 0;
    RenderCommand::SetInline32bitDataGraphics(0, &d, 1);
}
```

If instance data (data) is provided, it copies the instance data to the InstanceDataBuffer at an offset determined by m_InstanceDataForks. It also increments the m_InstanceDataForks counter to track how many instance data blocks have been processed.
If no instance data is provided, it sets a default value (0) to indicate no specific instance data for the current draw call.

Setting Inline Resource Views

```
RenderCommand::SetInlineResourceViewGraphics(1, TriangleDataBuffer.Get().get(), cbvdesc, ViewAccessType_ConstantBuffer);
RenderCommand::SetInlineResourceViewGraphics(2, InstanceDataBuffer.Get().get(), sbvdesc, ViewAccessType_GpuRead);
RenderCommand::SetInlineResourceViewGraphics(3, LightDataBuffer.Get().get(), lightsbvdesc, ViewAccessType_GpuRead);
```

These lines set up the resource views for different buffers:
TriangleDataBuffer: The vertex and index data.
InstanceDataBuffer: The instance-specific data.
LightDataBuffer: The light data used for shading or lighting effects.

The data is passed with different access types, such as ConstantBuffer and GpuRead, which define how the GPU will access these resources.

Setting Additional Resource Views

```
DrawVBIB_SetInlineResourceViewGraphics();
```

This function call is used to handle additional resource views (related to post-processing stages of the pipeline).

Setting Buffers and Primitive Topology

```
RenderCommand::SetIndexBuffer(ib);
RenderCommand::SetVertexBuffer(vb, 1);
```

```
RenderCommand::SetPrimitiveTopology(Topology);

Sets the index buffer (ib), vertex buffer (vb), and the primitive topology (Topology). The topology defines the way
vertices are interpreted, such as TriangleList, LineStrip, etc.

-----------------------------------------------------------------------------------------------------
Draw Indexed Command

RenderCommand::DrawIndexed(0, 0, ib->m_IndexCount);

This is the main draw command that issues a draw call using indexed geometry. It uses the index count from the provided
IndexBufferView (ib) to determine how many indices to process.

-----------------------------------------------------------------------------------------------------
Visualizing Normals

if (m_VisualiseNormals)
{
    RenderCommand::SetPipelineState(NormalVisualizer.get());
    RenderCommand::DrawIndexed(0, 0, ib->m_IndexCount);
    RenderCommand::SetPipelineState(Pipelines[CURRENT_STATE_].get());
}

If normal visualization is enabled (m_VisualiseNormals), it switches the pipeline state to a "normal visualizer" state
and issues another draw call. This is often used in debugging to visualize how normals are applied to the geometry.
After visualizing the normals, the pipeline state is reset to the current state to continue with the regular rendering
process.

-----------------------------------------------------------------------------------------------------
Output

This function sets up the necessary resources, instance data, and pipeline states to render geometry using vertex and
index buffers. It also optionally handles normal visualization for debugging or visualization purposes.
The function does not return any value but directly interacts with the GPU to manage rendering.

-----------------------------------------------------------------------------------------------------
Conceptual Summary
The DrawVBIB() function is a critical part of the rendering pipeline, responsible for configuring resources and issuing
draw commands. It checks if the pipeline state is dirty and ensures that all necessary resources are set up for
rendering. It also provides flexibility for instance-based rendering and optional debugging features, like normal
visualization.
```

```
void PennyRender::renderScreen(Texture renderTexture) {

    SetRenderTargets_(OutputTexture,OutputRTV);

    //RenderCommand::ClearRenderTargets(OutputRTVs_0.Get().get(), 1, float4(0, 0, 0, 0));

    //rendering scene
    Begin();//

    uint32_t Indexdata[]
    {
        0,1,2
    };

    std::vector<pen::Vertex> movedvertices = vertices_screen;

    pen::InstanceData testdata;

    testdata.TextureIndex = 10;

    for (auto& a : movedvertices) { a.Position += float3(2.5, 0, 0); }
    DrawVertices(movedvertices.data(), 3, Indexdata, 3, &testdata);



    pen::Vertex Quad[4] =
    {
        {float3(-1,1,0),float3(0,0,0),float2(0,0),    float3(0,0,-1)    },
        {float3(-1,-1,0),float3(0,0,0),float2(0,1),    float3(0,0,-1)    },
```

```
        {float3(1,-1,0),float3(0,0,0),float2(1,1),    float3(0,0,-1)    },
        {float3(1,1,0),float3(0,0,0),float2(1,0),    float3(0,0,-1)    }
    };

    uint32_t quadindices[6] =
    {
        0,1,2,2,3,0
    };

    testdata.TextureIndex = renderTexture->GetViewIndex();

    matrix World = glm::translate(float3(-5, 0, 0));
    testdata.World = World * glm::translate(float3(5, 0, -5)) * (90.f, 1.f, 1.f) * glm::scale(float3(9, 5, 1)); //find
out proper window size later (maybe ask naman if he knows?)
    testdata.ColourScale = float4(1, 1, 1, 0.9);
    testdata.SpecularColor = SpecularC;
    testdata.SpecularPower = SpecularPower;
    testdata.SpecularWeight = SpecularWeight;

    DrawVertices(Quad, 4, quadindices, 6, &testdata);



    End();

    //end of render

    GraphicsContext::SetRenderTargetToBackBuffer();

}
```

```
PennyRender::renderScreen()
-----------------------------------------------------------------------------------------------------------------------
Purpose
-----------------------------------------------------------------------------------------------------------------------

The renderScreen() function is responsible for rendering the scene onto a texture and then drawing a quad to the screen,
using the specified Texture as the source for the quad. It sets up the necessary render targets, handles the rendering of
vertices and index data, and applies instance-specific transformation and texture data before completing the rendering
process.

-----------------------------------------------------------------------------------------------------------------------
Function Overview
-----------------------------------------------------------------------------------------------------------------------
Setting Render Targets

SetRenderTargets_(OutputTexture, OutputRTV);

This function call sets the render targets for the current rendering pass. OutputTexture is the texture where the scene
will be rendered, and OutputRTV is the render target view used for that texture.

-----------------------------------------------------------------------------------------------------------------------
Beginning the Render

Begin();

This function call starts the rendering process by preparing necessary buffers and setting up the pipeline. It’s a
prerequisite for rendering any geometry.

-----------------------------------------------------------------------------------------------------------------------
Drawing Triangular Vertices

uint32_t Indexdata[] { 0, 1, 2 };
std::vector<pen::Vertex> movedvertices = vertices_screen;
pen::InstanceData testdata;
testdata.TextureIndex = 10;


for (auto& a : movedvertices) { a.Position += float3(2.5, 0, 0); }
DrawVertices(movedvertices.data(), 3, Indexdata, 3, &testdata);
```

First, a triangle is defined using Indexdata[] and movedvertices[] (the vertices of a triangle). The vertices are moved along the X-axis by a value of 2.5 for each vertex in movedvertices.
Then, DrawVertices() is called to render the triangle with the provided instance data (testdata). The TextureIndex is set to 10 for the texture that will be used during rendering.

Defining Quad Vertices for Full-Screen Rendering

```
pen::Vertex Quad[4] = {
    {float3(-1,1,0),float3(0,0,0),float2(0,0), float3(0,0,-1)},
    {float3(-1,-1,0),float3(0,0,0),float2(0,1), float3(0,0,-1)},
    {float3(1,-1,0),float3(0,0,0),float2(1,1), float3(0,0,-1)},
    {float3(1,1,0),float3(0,0,0),float2(1,0), float3(0,0,-1)}
};
```

Defines the vertices of a quad, which will be rendered as a full-screen object. The quad has four corners, with each vertex having a position (float3), a texture coordinate (float2), and a normal vector (float3).

Defining Quad Indices

```
uint32_t quadindices[6] = { 0, 1, 2, 2, 3, 0 };
```

Defines the indices for the quad. The indices refer to the vertices in Quad[] and determine how the vertices are connected to form two triangles (one for each half of the quad).

Setting Instance Data for Quad

```
testdata.TextureIndex = renderTexture->GetViewIndex();
matrix World = glm::translate(float3(-5, 0, 0));
testdata.World = World * glm::translate(float3(5, 0, -5)) * (90.f, 1.f, 1.f) * glm::scale(float3(9, 5, 1));
testdata.ColourScale = float4(1, 1, 1, 0.9);
testdata.SpecularColor = SpecularC;
testdata.SpecularPower = SpecularPower;
testdata.SpecularWeight = SpecularWeight;
```

Sets the instance data (testdata) for the quad:
TextureIndex: Uses the texture's view index (renderTexture->GetViewIndex()).
World: Applies a series of transformations, including translation, rotation, and scaling to the quad.
ColourScale, SpecularColor, SpecularPower, and SpecularWeight: Additional data for shading, lighting, and material properties (such as color scaling and specular effects).

Drawing the Quad

```
DrawVertices(Quad, 4, quadindices, 6, &testdata);
```

This call renders the quad using the vertices (Quad), indices (quadindices), and the instance data (testdata). The quad is drawn to the screen, using the texture specified in testdata.

Ending the Render

```
End();
```

This function call ends the rendering process. It handles any final updates to buffers, synchronizes resources, or finalizes the frame for output.

Setting the Render Target to the Back Buffer

```
GraphicsContext::SetRenderTargetToBackBuffer();
```

After the custom rendering is completed, this function call restores the render target back to the back buffer (i.e., the final screen output), preparing for the next frame or draw call.

Output

This function performs rendering to a texture and draws the resulting texture as a full-screen quad. It allows for advanced rendering effects, such as post-processing or offscreen rendering, followed by displaying the rendered texture. The output is rendered on the back buffer (screen), completing the frame's visual output.

Conceptual Summary

The renderScreen() function is designed to facilitate offscreen rendering onto a texture and then display that texture on the screen as a quad. This is useful for various advanced graphics techniques like post-processing, shadow mapping, or screen-space effects. It efficiently manages the render targets, applies transformations, and draws both simple geometry and textured surfaces.

```
void PennyRender::MultiRenderPass(Texture renderTexture) { // not working


/*    renderScreenTest(renderTexture, OutputTextures_[0], OutputRTVs_[0], OutputSRVs_[0]);
    Texture tex = OutputSRVs_[0].Get();
    renderToScreen_end(OutputRTVs_[0], OutputSRVs_[0]);


    renderScreenTest(renderTexture, OutputTextures_1, OutputRTVs_1, OutputSRVs_1);*/
    renderScreenPass(renderTexture, OutputTextures_0, OutputRTVs_0, OutputSRVs_0); //
    Texture tex = OutputSRVs_0.Get();
    renderToScreen_end(OutputRTVs_0, OutputSRVs_0);
    renderScreen(tex);//always has to be the last one (as its "texture" -SRV- is what gets pasted through to the preive
screen)
    //renderToScreen_end(OutputRTVs_0, OutputSRVs_0);


}
```

PennyRender::MultiRenderPass()

Purpose

The MultiRenderPass() function is intended to execute multiple rendering passes in sequence. It handles rendering to offscreen textures, followed by final rendering to the screen. Although currently commented out and non-functional, the general idea is to process different effects or steps in multiple passes, updating the output after each pass, and finally displaying the result.

Function Overview

Rendering the First Pass to an Offscreen Texture

```
renderScreenPass(renderTexture, OutputTextures_0, OutputRTVs_0, OutputSRVs_0);
```

This line calls renderScreenPass() to perform the first render pass. The pass renders the scene to the texture specified by OutputTextures_0, using the corresponding render target view (OutputRTVs_0) and shader resource view (OutputSRVs_0). This is part of an offscreen rendering technique, for post-processing effects or intermediate results.

Getting the Texture for Further Processing

```
Texture tex = OutputSRVs_0.Get();
```

After completing the first render pass, this line retrieves the texture from OutputSRVs_0 (the shader resource view) that contains the output of the first render pass. This texture will be used in subsequent passes or for display.

Finalizing the Render to Screen

```
renderToScreen_end(OutputRTVs_0, OutputSRVs_0);
```

This function call, renderToScreen_end(), finalizes the rendering for the current pass by updating the render target views and shader resource views. It effectively marks the end of the render pass and prepares the scene for displaying or processing further.

Rendering the Final Result to the Screen

```
renderScreen(tex);
```

This function call uses the texture (tex) from the previous render pass and renders it to the screen as a full-screen quad. This is usually the final step in the rendering pipeline when applying post-processing effects or displaying the final result.

Commented Out / Additional Render Passes

```
/*
renderScreenTest(renderTexture, OutputTextures_[0], OutputRTVs_[0], OutputSRVs_[0]);
Texture tex = OutputSRVs_[0].Get();
renderToScreen_end(OutputRTVs_[0], OutputSRVs_[0]);


renderScreenTest(renderTexture, OutputTextures_1, OutputRTVs_1, OutputSRVs_1);
```

```
*/

Purpose:
The commented-out code represents additional render passes that were planned but are currently not functional. These
passes would render to different textures and views (OutputTextures_[0], OutputTextures_1, etc.), and the result would
then be passed through the screen render process.
These lines suggest the possibility of handling multiple intermediate render passes (e.g., different visual effects or
steps).

-------------------------------------------------------------------------------------------
Output

Effect:
The function ensures that multiple render passes are performed and that the final rendered image is displayed on the
screen. The function first renders to an offscreen texture, then processes the result, and finally renders the processed
texture to the screen.
The code hints that the function was designed for handling multiple render passes, but it is currently incomplete or not
working.

-------------------------------------------------------------------------------------------
Conceptual Summary

The MultiRenderPass() function is designed to facilitate a multi-pass rendering pipeline where intermediate results are
rendered offscreen, processed, and then displayed. Although not fully functional in its current form, this pattern is
commonly used in rendering techniques like post-processing effects, shadow mapping, or deferred rendering, where each
pass serves a specific purpose, such as applying a filter or calculating lighting.
The function concludes with rendering the final processed texture to the screen.
```

```
void PennyRender::renderToScreen_end(MultiResource<RenderTargetViewRef> OutputRTV, MultiResource<TextureSRVRef>  OutputSRV) {
////next frame
++OutputSRV;
++OutputRTV;
}
```

PennyRender:: renderToScreen_end(MultiResource<RenderTargetViewRef> OutputRTV, MultiResource<TextureSRVRef> OutputSRV)

Purpose

The renderToScreen_end() function is used to finalize the process of rendering to the screen by updating the render target and shader resource view (SRV) for the next frame.

Function Overview

Incrementing the Render Target View (RTV)

```
++OutputRTV;
```

Purpose: This operation increments the Render Target View (RTV) to point to the next render target in the sequence.
Effect: This essentially prepares the system to render the next frame to a new target, moving forward in the sequence of render targets.

Incrementing the Shader Resource View (SRV)

```
++OutputSRV;
```

Purpose: Similar to the RTV, this operation increments the Shader Resource View (SRV) to the next texture in the sequence.
Effect: It ensures that the next texture resource (for example, a texture containing the result of the current frame's rendering) is used in subsequent shader operations, typically for post-processing or any other shader-based effects.

Conceptual Summary

The renderToScreen_end() function serves as a mechanism to increment the render target and shader resource view pointers, effectively preparing them for the next frame. This allows the system to move from one frame's output to the next frame's input seamlessly, ensuring continuous rendering.

Drawing to the render target

```
void PennyRender::renderToScreen_init() {


    //--textDesc init
    TextureDesc textDesc{};


    textDesc.Format = DataFormat_DEFAULT_BACKBUFFER;
    textDesc.AccessFlags = ResourceAccessFlag_GpuRead | ResourceAccessFlag_GpuWrite;
    textDesc.Type = TextureType_RenderTarget;
    textDesc.Dimension = TextureDimension_Texture2D;
    textDesc.Width = GraphicsContext::GetClientWidth();
    textDesc.Height = GraphicsContext::GetClientHeight();
    textDesc.Depth = 1;
    textDesc.Mips = 1;
    textDesc.Clear = ClearValue(float4(0,0,0,0));


    OutputTexture.ForEach([&](auto& Texture) { Texture = ResourceFactory::CreateTextureBuffer(textDesc); });


    //create output view desc
    TextureViewDesc srv{};
    srv.Dimension = TextureDimension_Texture2D;
    srv.ArraySize = 1;
    srv.Format = DataFormat_DEFAULT_BACKBUFFER;
    srv.MipLevels = 1;
    srv.MipSlice = 0;


    //RenderTargetDesc rtv_desc{.MipSlice = 0, .PlaneSlice = 0};
    //OutputRTV.ForEach([&, i = 0](auto& view) mutable -> void {view = ResourceFactory::CreateRenderTargetView(&rtv_desc,
OutputTexture[i]); i++; });
    OutputRTV.ForEach([&, i = 0](auto& view) mutable -> void {view = ResourceFactory::CreateRenderTargetView(0,
OutputTexture[i]); i++; }); //the 0 here will cout some warning messages, but einar said to ignore it since it works and
doesnt negatively effect anything
    OutputSRV.ForEach([&, i = 0](auto& view) mutable -> void {view =
ResourceFactory::CreateTextureShaderResourceView(srv, OutputTexture[i]); i++; });



}
```

```
PennyRender::renderToScreen_init
-----------------------------------------------------------------------------------------
Purpose

Initializes the output render targets and shader resource views used for rendering the final scene image to the screen.
This setup supports post-processing, multi-pass rendering, and general output-to-texture workflows.

-----------------------------------------------------------------------------------------
Function Overview
-----------------------------------------------------------------------------------------
Texture Descriptor Setup

TextureDesc textDesc{};
textDesc.Format = DataFormat_DEFAULT_BACKBUFFER;
textDesc.AccessFlags = ResourceAccessFlag_GpuRead | ResourceAccessFlag_GpuWrite;
textDesc.Type = TextureType_RenderTarget;
...
textDesc.Width = GraphicsContext::GetClientWidth();
textDesc.Height = GraphicsContext::GetClientHeight();



 Describes a 2D texture used as a render target with GPU read/write access.
Clear colour:
 Set to transparent black (0,0,0,0) for clean slate rendering.

-----------------------------------------------------------------------------------------
Create Render Target Textures

OutputTexture.ForEach([&](auto& Texture) {
    Texture = ResourceFactory::CreateTextureBuffer(textDesc);
});

 Allocates the actual GPU texture resources based on textDesc — typically one for each render pass or buffer frame.
```

```
-------------------------------------------------------------------------------------------
Shader Resource View Descriptor Setup

TextureViewDesc srv{};
srv.Dimension = TextureDimension_Texture2D;
srv.ArraySize = 1;
srv.Format = DataFormat_DEFAULT_BACKBUFFER;
srv.MipLevels = 1;
srv.MipSlice = 0;


Defines how the texture will be accessed in shaders — as a 2D resource with one mip level.

-------------------------------------------------------------------------------------------
Create Render Target Views (RTV)

OutputRTV.ForEach([&, i = 0](auto& view) mutable {
    view = ResourceFactory::CreateRenderTargetView(0, OutputTexture[i]);
    i++;
});



 For each output texture, a corresponding render target view is created so it can be written to by the GPU.
Note:
 The hardcoded 0 might emit a warning (as noted in the comment), but is safe per developer comments.

-------------------------------------------------------------------------------------------
Create Shader Resource Views (SRV)

OutputSRV.ForEach([&, i = 0](auto& view) mutable {
    view = ResourceFactory::CreateTextureShaderResourceView(srv, OutputTexture[i]);
    i++;
});


 Creates a view that allows shaders to sample from the rendered texture (e.g., for post-processing).

-------------------------------------------------------------------------------------------
Conceptual Summary

This function:
Allocates GPU render targets at screen resolution.
Sets up render target views to render into.
Sets up shader resource views to read from.
It’s foundational to enabling post-processing and effects that render from one texture and display the result on screen
or in a subsequent pass.
```

```
void PennyRender::renderToScreen_init_pass(MultiResource<TextureBufferRef> OutputTexture,
MultiResource<RenderTargetViewRef> OutputRTV, MultiResource<TextureSRVRef>  OutputSRV)
{
    //--textDesc init
    TextureDesc textDesc{};
    //textDesc.Format = renderTexture.get()->GetFormat();
    textDesc.Format = DataFormat_DEFAULT_BACKBUFFER;
    textDesc.AccessFlags = ResourceAccessFlag_GpuRead | ResourceAccessFlag_GpuWrite;
    textDesc.Type = TextureType_RenderTarget;
    textDesc.Dimension = TextureDimension_Texture2D;
    //textDesc.Width = renderTexture.get()->GetWidth(); //textDesc.Height = renderTexture.get()->GetHeight();
    textDesc.Width = GraphicsContext::GetClientWidth();
    textDesc.Height = GraphicsContext::GetClientHeight();
    textDesc.Depth = 1;
    textDesc.Mips = 1;
    textDesc.Clear = ClearValue(float4(0, 0, 0, 0));


    OutputTexture.ForEach([&](auto& Texture) { Texture = ResourceFactory::CreateTextureBuffer(textDesc); });


    //create srv disc
    TextureViewDesc srv{};
    srv.Dimension = TextureDimension_Texture2D;
    srv.ArraySize = 1;
    srv.Format = DataFormat_DEFAULT_BACKBUFFER;
    srv.MipLevels = 1;
```

```
    srv.MipSlice = 0;


    RenderTargetDesc rtv_desc{ .MipSlice = 0, .PlaneSlice = 0 };


    //OutputRTV.ForEach([&, i = 0](auto& view) mutable -> void {view = ResourceFactory::CreateRenderTargetView(&rtv_desc,
OutputTexture[i]); i++; });
    OutputRTV.ForEach([&, i = 0](auto& view) mutable -> void {view = ResourceFactory::CreateRenderTargetView(0,
OutputTexture[i]); i++; });
    OutputSRV.ForEach([&, i = 0](auto& view) mutable -> void {view =
ResourceFactory::CreateTextureShaderResourceView(srv, OutputTexture[i]); i++; });


}
```

```
PennyRender::renderToScreen_init
-------------------------------------------------------------------------------------------
Purpose

Initializes the output render targets and shader resource views used for rendering the final scene image to the screen.
This setup supports post-processing, multi-pass rendering, and general output-to-texture workflows.

-------------------------------------------------------------------------------------------
Function Overview
-------------------------------------------------------------------------------------------
Texture Descriptor Setup

TextureDesc textDesc{};
textDesc.Format = DataFormat_DEFAULT_BACKBUFFER;
textDesc.AccessFlags = ResourceAccessFlag_GpuRead | ResourceAccessFlag_GpuWrite;
textDesc.Type = TextureType_RenderTarget;
...
textDesc.Width = GraphicsContext::GetClientWidth();
textDesc.Height = GraphicsContext::GetClientHeight();



 Describes a 2D texture used as a render target with GPU read/write access.
Clear colour:
 Set to transparent black (0,0,0,0) for clean slate rendering.

-------------------------------------------------------------------------------------------
Create Render Target Textures

OutputTexture.ForEach([&](auto& Texture) {
    Texture = ResourceFactory::CreateTextureBuffer(textDesc);
});

 Allocates the actual GPU texture resources based on textDesc — typically one for each render pass or buffer frame.

-------------------------------------------------------------------------------------------
Shader Resource View Descriptor Setup

TextureViewDesc srv{};
srv.Dimension = TextureDimension_Texture2D;
srv.ArraySize = 1;
srv.Format = DataFormat_DEFAULT_BACKBUFFER;
srv.MipLevels = 1;
srv.MipSlice = 0;


Defines how the texture will be accessed in shaders — as a 2D resource with one mip level.

-------------------------------------------------------------------------------------------
Create Render Target Views (RTV)

OutputRTV.ForEach([&, i = 0](auto& view) mutable {
    view = ResourceFactory::CreateRenderTargetView(0, OutputTexture[i]);
    i++;
});



 For each output texture, a corresponding render target view is created so it can be written to by the GPU.
Note:
 The hardcoded 0 might emit a warning (as noted in the comment), but is safe per developer comments.
```

```
--------------------------------------------------------------------------------------------
Create Shader Resource Views (SRV)

OutputSRV.ForEach([&, i = 0](auto& view) mutable {
    view = ResourceFactory::CreateTextureShaderResourceView(srv, OutputTexture[i]);
    i++;
});


 Creates a view that allows shaders to sample from the rendered texture (e.g., for post-processing).

--------------------------------------------------------------------------------------------
Conceptual Summary

This function:
Allocates GPU render targets at screen resolution.
Sets up render target views to render into.
Sets up shader resource views to read from.
It’s foundational to enabling post-processing and effects that render from one texture and display the result on screen
or in a subsequent pass.
```

```
void PennyRender::renderScreen(Texture renderTexture) {


    SetRenderTargets_(OutputTexture,OutputRTV);


    //RenderCommand::ClearRenderTargets(OutputRTVs_0.Get().get(), 1, float4(0, 0, 0, 0));


    //rendering scene
    Begin();//


    uint32_t Indexdata[]
    {
        0,1,2
    };


    std::vector<pen::Vertex> movedvertices = vertices_screen;


    pen::InstanceData testdata;


    testdata.TextureIndex = 10;


    for (auto& a : movedvertices) { a.Position += float3(2.5, 0, 0); }
    DrawVertices(movedvertices.data(), 3, Indexdata, 3, &testdata);



    pen::Vertex Quad[4] =
    {
        {float3(-1,1,0),float3(0,0,0),float2(0,0),    float3(0,0,-1)    },
        {float3(-1,-1,0),float3(0,0,0),float2(0,1),    float3(0,0,-1)    },
        {float3(1,-1,0),float3(0,0,0),float2(1,1),    float3(0,0,-1)    },
        {float3(1,1,0),float3(0,0,0),float2(1,0),    float3(0,0,-1)    }
    };


    uint32_t quadindices[6] =
    {
        0,1,2,2,3,0
    };


    testdata.TextureIndex = renderTexture->GetViewIndex();


    matrix World = glm::translate(float3(-5, 0, 0));
    testdata.World = World * glm::translate(float3(5, 0, -5)) * (90.f, 1.f, 1.f) * glm::scale(float3(9, 5, 1)); //find
out proper window size later (maybe ask naman if he knows?)
    testdata.ColourScale = float4(1, 1, 1, 0.9);
    testdata.SpecularColor = SpecularC;
    testdata.SpecularPower = SpecularPower;
    testdata.SpecularWeight = SpecularWeight;
```

```
    DrawVertices(Quad, 4, quadindices, 6, &testdata);

    //current_instancedata = testdata;///this is where we get data for the multiple passes

    End();

    //end of render

    GraphicsContext::SetRenderTargetToBackBuffer();

}
```

PennyRender::renderScreen()

Purpose

The renderScreen() function is responsible for rendering the scene onto a texture and then drawing a quad to the screen, using the specified Texture as the source for the quad. It sets up the necessary render targets, handles the rendering of vertices and index data, and applies instance-specific transformation and texture data before completing the rendering process.

Function Overview

Setting Render Targets

```
SetRenderTargets_(OutputTexture, OutputRTV);
```

This function call sets the render targets for the current rendering pass. OutputTexture is the texture where the scene will be rendered, and OutputRTV is the render target view used for that texture.

Beginning the Render

```
Begin();
```

This function call starts the rendering process by preparing necessary buffers and setting up the pipeline. It’s a prerequisite for rendering any geometry.

Drawing Triangular Vertices

```
uint32_t Indexdata[] { 0, 1, 2 };
std::vector<pen::Vertex> movedvertices = vertices_screen;
pen::InstanceData testdata;
testdata.TextureIndex = 10;


for (auto& a : movedvertices) { a.Position += float3(2.5, 0, 0); }
DrawVertices(movedvertices.data(), 3, Indexdata, 3, &testdata);
```

First, a triangle is defined using Indexdata[] and movedvertices[] (the vertices of a triangle). The vertices are moved along the X-axis by a value of 2.5 for each vertex in movedvertices.
Then, DrawVertices() is called to render the triangle with the provided instance data (testdata). The TextureIndex is set to 10 for the texture that will be used during rendering.

Defining Quad Vertices for Full-Screen Rendering

```
pen::Vertex Quad[4] = {
    {float3(-1,1,0),float3(0,0,0),float2(0,0), float3(0,0,-1)},
    {float3(-1,-1,0),float3(0,0,0),float2(0,1), float3(0,0,-1)},
    {float3(1,-1,0),float3(0,0,0),float2(1,1), float3(0,0,-1)},
    {float3(1,1,0),float3(0,0,0),float2(1,0), float3(0,0,-1)}
};
```

Defines the vertices of a quad, which will be rendered as a full-screen object. The quad has four corners, with each vertex having a position (float3), a texture coordinate (float2), and a normal vector (float3).

Defining Quad Indices

```
uint32_t quadindices[6] = { 0, 1, 2, 2, 3, 0 };
```

Defines the indices for the quad. The indices refer to the vertices in Quad[] and determine how the vertices are connected to form two triangles (one for each half of the quad).

Setting Instance Data for Quad

```
testdata.TextureIndex = renderTexture->GetViewIndex();
matrix World = glm::translate(float3(-5, 0, 0));
testdata.World = World * glm::translate(float3(5, 0, -5)) * (90.f, 1.f, 1.f) * glm::scale(float3(9, 5, 1));
testdata.ColourScale = float4(1, 1, 1, 0.9);
testdata.SpecularColor = SpecularC;
testdata.SpecularPower = SpecularPower;
testdata.SpecularWeight = SpecularWeight;
```

Sets the instance data (testdata) for the quad:
TextureIndex: Uses the texture's view index (renderTexture->GetViewIndex()).
World: Applies a series of transformations, including translation, rotation, and scaling to the quad.
ColourScale, SpecularColor, SpecularPower, and SpecularWeight: Additional data for shading, lighting, and material properties (such as color scaling and specular effects).

Drawing the Quad

```
DrawVertices(Quad, 4, quadindices, 6, &testdata);
```

This call renders the quad using the vertices (Quad), indices (quadindices), and the instance data (testdata). The quad is drawn to the screen, using the texture specified in testdata.

Ending the Render

```
End();
```

This function call ends the rendering process. It handles any final updates to buffers, synchronizes resources, or finalizes the frame for output.

Setting the Render Target to the Back Buffer

```
GraphicsContext::SetRenderTargetToBackBuffer();
```

After the custom rendering is completed, this function call restores the render target back to the back buffer (i.e., the final screen output), preparing for the next frame or draw call.

Output

This function performs rendering to a texture and draws the resulting texture as a full-screen quad. It allows for advanced rendering effects, such as post-processing or offscreen rendering, followed by displaying the rendered texture. The output is rendered on the back buffer (screen), completing the frame's visual output.

Conceptual Summary

The renderScreen() function is designed to facilitate offscreen rendering onto a texture and then display that texture on the screen as a quad. This is useful for various advanced graphics techniques like post-processing, shadow mapping, or screen-space effects. It efficiently manages the render targets, applies transformations, and draws both simple geometry and textured surfaces.

```
   void SetRenderTargets_(MultiResource<TextureBufferRef> OutputTexture, MultiResource<RenderTargetViewRef> OutputRTV) {


        TextureBarrier renderTargetBarrier{};
        renderTargetBarrier.SyncBefore = SKTBD_SYNC_PIXEL_SHADING;
        renderTargetBarrier.SyncAfter = SKTBD_SYNC_RENDER_TARGET;
        renderTargetBarrier.AccessBefore = SKTBD_ACCESS_COMMON;
        renderTargetBarrier.AccessAfter = SKTBD_ACCESS_RENDER_TARGET;
        renderTargetBarrier.LayoutBefore = SKTBD_LAYOUT_COMMON;
        renderTargetBarrier.LayoutAfter = SKTBD_LAYOUT_RENDER_TARGET;
        renderTargetBarrier.Resource = OutputTexture.Get().get();//
        renderTargetBarrier.SubresourceRange = TextureSubresourceRange();
```

```
        BarrierGroup grouptoRTV(&renderTargetBarrier);
        RenderCommand::Barrier(&grouptoRTV, 1);


        RenderCommand::SetRenderTargets(OutputRTV.Get().get(), 1, GraphicsContext::GetDefaultDepthBuffer());



    }
```

```
void SetRenderTargets_(MultiResource<TextureBufferRef> OutputTexture, MultiResource<RenderTargetViewRef> OutputRTV)
-----------------------------------------------------------------------------------------------------------
Purpose

Prepares and sets the render target for rendering by:
Transitioning the texture resource to a render-target-compatible state using a barrier.
Binding the appropriate render target view (RTV) and depth buffer to the graphics pipeline.
This function ensures that the GPU is synchronized and ready to render into the target texture.
-----------------------------------------------------------------------------------------------------------
Function Overview
-----------------------------------------------------------------------------------------------------------
Define Texture Barrier for Layout Transition

TextureBarrier renderTargetBarrier{};
renderTargetBarrier.SyncBefore = SKTBD_SYNC_PIXEL_SHADING;
renderTargetBarrier.SyncAfter = SKTBD_SYNC_RENDER_TARGET;
renderTargetBarrier.AccessBefore = SKTBD_ACCESS_COMMON;
renderTargetBarrier.AccessAfter = SKTBD_ACCESS_RENDER_TARGET;
renderTargetBarrier.LayoutBefore = SKTBD_LAYOUT_COMMON;
renderTargetBarrier.LayoutAfter = SKTBD_LAYOUT_RENDER_TARGET;
renderTargetBarrier.Resource = OutputTexture.Get().get();


 Ensures the output texture is transitioned from a general/common state to a state suitable for render target usage.
Why It’s Important:
 DirectX 12-style explicit graphics APIs require manual state transitions to ensure correct GPU access behavior.

-----------------------------------------------------------------------------------------------------------
Apply the Barrier

BarrierGroup grouptoRTV(&renderTargetBarrier);
RenderCommand::Barrier(&grouptoRTV, 1);


 Wraps the barrier into a group and submits it, forcing the transition on the GPU before the next draw calls.

-----------------------------------------------------------------------------------------------------------
Set the Render Target and Depth Buffer

RenderCommand::SetRenderTargets(OutputRTV.Get().get(), 1, GraphicsContext::GetDefaultDepthBuffer());

Binds the specified render target (RTV) and the default depth buffer as active targets for rendering.

-----------------------------------------------------------------------------------------------------------
Conceptual Summary

This function:
Ensures the render target texture is in the right GPU state.
Binds it for rendering.
Enables subsequent draw calls to properly output to the desired texture.
It’s a critical utility for frame rendering, post-processing passes, and any render-to-texture workflows.
```

```
void PennyRender::MultiRenderPass(Texture renderTexture) { // not working


/*    renderScreenTest(renderTexture, OutputTextures_[0], OutputRTVs_[0], OutputSRVs_[0]);
    Texture tex = OutputSRVs_[0].Get();
    renderToScreen_end(OutputRTVs_[0], OutputSRVs_[0]);


    renderScreenTest(renderTexture, OutputTextures_1, OutputRTVs_1, OutputSRVs_1);*/
    renderScreenPass(renderTexture, OutputTextures_0, OutputRTVs_0, OutputSRVs_0); //
    Texture tex = OutputSRVs_0.Get();
    renderToScreen_end(OutputRTVs_0, OutputSRVs_0);
    renderScreen(tex);//always has to be the last one (as its "texture" -SRV- is what gets pasted through to the preive
screen)
```

```
    //renderToScreen_end(OutputRTVs_0, OutputSRVs_0);

}
```

PennyRender::MultiRenderPass()

Purpose

The MultiRenderPass() function is intended to execute multiple rendering passes in sequence. It handles rendering to offscreen textures, followed by final rendering to the screen. Although currently commented out and non-functional, the general idea is to process different effects or steps in multiple passes, updating the output after each pass, and finally displaying the result.

Function Overview

Rendering the First Pass to an Offscreen Texture

```
renderScreenPass(renderTexture, OutputTextures_0, OutputRTVs_0, OutputSRVs_0);
```

This line calls renderScreenPass() to perform the first render pass. The pass renders the scene to the texture specified by OutputTextures_0, using the corresponding render target view (OutputRTVs_0) and shader resource view (OutputSRVs_0). This is part of an offscreen rendering technique, for post-processing effects or intermediate results.

Getting the Texture for Further Processing

```
Texture tex = OutputSRVs_0.Get();
```

After completing the first render pass, this line retrieves the texture from OutputSRVs_0 (the shader resource view) that contains the output of the first render pass. This texture will be used in subsequent passes or for display.

Finalizing the Render to Screen

```
renderToScreen_end(OutputRTVs_0, OutputSRVs_0);
```

This function call, renderToScreen_end(), finalizes the rendering for the current pass by updating the render target views and shader resource views. It effectively marks the end of the render pass and prepares the scene for displaying or processing further.

Rendering the Final Result to the Screen

```
renderScreen(tex);
```

This function call uses the texture (tex) from the previous render pass and renders it to the screen as a full-screen quad. This is usually the final step in the rendering pipeline when applying post-processing effects or displaying the final result.

Commented Out / Additional Render Passes

```
/*
renderScreenTest(renderTexture, OutputTextures_[0], OutputRTVs_[0], OutputSRVs_[0]);
Texture tex = OutputSRVs_[0].Get();
renderToScreen_end(OutputRTVs_[0], OutputSRVs_[0]);


renderScreenTest(renderTexture, OutputTextures_1, OutputRTVs_1, OutputSRVs_1);
*/
```

Purpose:
The commented-out code represents additional render passes that were planned but are currently not functional. These passes would render to different textures and views (OutputTextures_[0], OutputTextures_1, etc.), and the result would then be passed through the screen render process.
These lines suggest the possibility of handling multiple intermediate render passes (e.g., different visual effects or steps).

Output

Effect:
The function ensures that multiple render passes are performed and that the final rendered image is displayed on the screen. The function first renders to an offscreen texture, then processes the result, and finally renders the processed texture to the screen.

The code hints that the function was designed for handling multiple render passes, but it is currently incomplete or not working.

Conceptual Summary

The MultiRenderPass() function is designed to facilitate a multi-pass rendering pipeline where intermediate results are rendered offscreen, processed, and then displayed. Although not fully functional in its current form, this pattern is commonly used in rendering techniques like post-processing effects, shadow mapping, or deferred rendering, where each pass serves a specific purpose, such as applying a filter or calculating lighting.
The function concludes with rendering the final processed texture to the screen.

```
void PennyRender::renderToScreen_end(MultiResource<RenderTargetViewRef> OutputRTV, MultiResource<TextureSRVRef>  OutputSRV) {
////next frame
++OutputSRV;
++OutputRTV;
}
```

PennyRender:: renderToScreen_end(MultiResource<RenderTargetViewRef> OutputRTV, MultiResource<TextureSRVRef> OutputSRV)

Purpose

The renderToScreen_end() function is used to finalize the process of rendering to the screen by updating the render target and shader resource view (SRV) for the next frame.

Function Overview

Incrementing the Render Target View (RTV)

```
++OutputRTV;
```

Purpose: This operation increments the Render Target View (RTV) to point to the next render target in the sequence.
Effect: This essentially prepares the system to render the next frame to a new target, moving forward in the sequence of render targets.

Incrementing the Shader Resource View (SRV)

```
++OutputSRV;
```

Purpose: Similar to the RTV, this operation increments the Shader Resource View (SRV) to the next texture in the sequence.
Effect: It ensures that the next texture resource (for example, a texture containing the result of the current frame's rendering) is used in subsequent shader operations, typically for post-processing or any other shader-based effects.

Conceptual Summary

The renderToScreen_end() function serves as a mechanism to increment the render target and shader resource view pointers, effectively preparing them for the next frame. This allows the system to move from one frame's output to the next frame's input seamlessly, ensuring continuous rendering.

```
void PennyRender::PennyBoardIMGUI_PreveiwScreen() {
    //-- barriers to tye/sync render target
    TextureBarrier renderTargetBarrier{};
    renderTargetBarrier.SyncBefore = SKTBD_SYNC_RENDER_TARGET;
    renderTargetBarrier.SyncAfter = SKTBD_SYNC_PIXEL_SHADING;
    renderTargetBarrier.AccessBefore = SKTBD_ACCESS_RENDER_TARGET;
    renderTargetBarrier.AccessAfter = SKTBD_ACCESS_COMMON;
    renderTargetBarrier.LayoutBefore = SKTBD_LAYOUT_RENDER_TARGET;
    renderTargetBarrier.LayoutAfter = SKTBD_LAYOUT_COMMON;
    renderTargetBarrier.Resource = OutputTexture.Get().get();
    renderTargetBarrier.SubresourceRange = TextureSubresourceRange();//


    BarrierGroup grouptoRTV(&renderTargetBarrier);
    RenderCommand::Barrier(&grouptoRTV, 1);
    //--


    ImGui::SeparatorText("Preveiw screen");


    ImGui::SliderInt("Screen size", &previewscreenSize, 1, 4, "Screen size / %d%");
    ImGui::SetItemTooltip("Preveiw screen size, Fullsize/ n");
```

```
    //preveiw screen
    ImGuiIO& io = ImGui::GetIO();
    ImTextureID my_tex_id = OutputSRV.Get()->GetImTextureID();//turning srv to a usable ImTextureID - for preview screen
    //move to next frame
    renderToScreen_end(OutputRTV, OutputSRV);


    float my_tex_w = (float)1366 / previewscreenSize;
    float my_tex_h = (float)768 / previewscreenSize;
    {
        static bool use_text_color_for_tint = false;


        ImVec2 uv_min = ImVec2(0.0f, 0.0f);                 // Top-left
        ImVec2 uv_max = ImVec2(1.0f, 1.0f);                 // Lower-right
        ImVec4 tint_col = use_text_color_for_tint ? ImGui::GetStyleColorVec4(ImGuiCol_Text) : ImVec4(1.0f, 1.0f, 1.0f,
1.0f);   // No tint
        ImVec4 border_col = ImGui::GetStyleColorVec4(ImGuiCol_Border);
        ImGui::Image(my_tex_id, ImVec2(my_tex_w, my_tex_h), uv_min, uv_max, tint_col, border_col);
        ImGui::SetItemTooltip("Preveiw screen;\nshow the effected in development before adding to the proper
screen/view");
    }
```

```
PennyRender::PennyBoardIMGUI_PreveiwScreen()
-------------------------------------------------------------------------------------------------------------------
Purpose

This function prepares and displays a preview of the final rendered output texture inside an ImGui window. It ensures
proper GPU synchronization (using a barrier) before showing the texture, and provides basic UI controls to scale the
preview.

-------------------------------------------------------------------------------------------------------------------
Barrier: Transition the Render Target to Common State

TextureBarrier renderTargetBarrier{};
renderTargetBarrier.SyncBefore = SKTBD_SYNC_RENDER_TARGET;
renderTargetBarrier.SyncAfter = SKTBD_SYNC_PIXEL_SHADING;
renderTargetBarrier.AccessBefore = SKTBD_ACCESS_RENDER_TARGET;
renderTargetBarrier.AccessAfter = SKTBD_ACCESS_COMMON;
renderTargetBarrier.LayoutBefore = SKTBD_LAYOUT_RENDER_TARGET;
renderTargetBarrier.LayoutAfter = SKTBD_LAYOUT_COMMON;
renderTargetBarrier.Resource = OutputTexture.Get().get();
renderTargetBarrier.SubresourceRange = TextureSubresourceRange();
BarrierGroup grouptoRTV(&renderTargetBarrier);
RenderCommand::Barrier(&grouptoRTV, 1);

Ensures the output texture is no longer being written to as a render target and is now in a state where it can be read as
a shader resource for preview.

-------------------------------------------------------------------------------------------------------------------
ImGui UI Controls and Preview Header

ImGui::SeparatorText("Preveiw screen");
ImGui::SliderInt("Screen size", &previewscreenSize, 1, 4, "Screen size / %d%");
ImGui::SetItemTooltip("Preveiw screen size, Fullsize/ n");

Provides a user interface to scale the preview size using a slider.
previewscreenSize controls how large the preview appears relative to full size (1 = full size, 4 = quarter size).

-------------------------------------------------------------------------------------------------------------------
Get Texture ID and Advance Frame

ImGuiIO& io = ImGui::GetIO();
ImTextureID my_tex_id = OutputSRV.Get()->GetImTextureID(); // Get a usable texture ID for ImGui
renderToScreen_end(OutputRTV, OutputSRV); // Advance resource pointer to next frame

Converts the Shader Resource View into a format (ImTextureID) usable by ImGui for image display.
Also advances to the next resource in the MultiResource pool to handle double/triple buffering.

-------------------------------------------------------------------------------------------------------------------
Display the Texture as an ImGui Image

float my_tex_w = (float)1366 / previewscreenSize;
float my_tex_h = (float)768 / previewscreenSize;
```

```
ImVec2 uv_min = ImVec2(0.0f, 0.0f); // Top-left
ImVec2 uv_max = ImVec2(1.0f, 1.0f); // Bottom-right
ImVec4 tint_col = ImVec4(1.0f, 1.0f, 1.0f, 1.0f); // No tint
ImVec4 border_col = ImGui::GetStyleColorVec4(ImGuiCol_Border);


ImGui::Image(my_tex_id, ImVec2(my_tex_w, my_tex_h), uv_min, uv_max, tint_col, border_col);
ImGui::SetItemTooltip("Preveiw screen;\nshow the effected in development before adding to the proper screen/view");

Displays the rendered texture inside the ImGui UI window as a resizable image.
Uses ImGui::Image() to show the texture and adds a helpful tooltip explaining the purpose of the preview screen.

-----------------------------------------------------------------------------------------------------------
Conceptual Summary

Ensures GPU synchronization before accessing the texture as an SRV.
Provides a simple and flexible UI element in ImGui to preview the rendering output.
Helps developers test and debug visual effects in isolation before applying them to the final screen output.
```

Misc Functions

```
// get / set fun
InstanceData getCurrentInstanceData() { return current_instancedata; }
void setCurrentInstanceData(InstanceData current_data) { current_instancedata = current_data; }

MultiResource<TextureBufferRef> getCurrentOutputTexture() { return OutputTexture; }
MultiResource<TextureSRVRef> getCurrentOutputSRVData() { return OutputSRV; }
MultiResource<RenderTargetViewRef> getCurrentOutputRTVData() { return OutputRTV; }
```

```
//effects - list stuff
std::vector < SavedEffect> savedEffect_List; // max4 for now

// Function to add an item to the end of the list
void addEffectToList(std::vector<SavedEffect>& list, SavedEffect effect) {
	list.push_back(effect);
}

void removeEffect(std::vector<SavedEffect>& list, size_t index) {
	if (index < list.size()) {
		list.erase(list.begin() + index); // Remove item and shift others up
	}
	else {
		std::cout << "Not vaild Index\n"; // have a pop-up layter
	}
}
```

```
void save_SavedEffect(SavedEffect effect) {addEffectToList(savedEffect_List, effect);}
SavedEffect ConvertToSavedEffect(const char* SavedName , ShaderToggles EffectInUse, Screen
{
	SavedEffect effect;

	effect.SavedName = SavedName;
	effect.EffectInUse = EffectInUse;

	effect.screen_buff = screen_buff;
	effect.scanline_buff = scanline_buff;
	effect.vignet_buff = vignet_buff;
	effect.posterize_buff = posterize_buff;
	effect.bloom_buff = bloom_buff;
	effect.bloom2_buff = bloom2_buff;
	effect.blurGauBuffer_buff = blurGauBuffer_buff;
	effect.chromaTrue_buff = chromaTrue_buff;
	effect.chromaStylized_buff = chromaStylized_buff;
	effect.colourGrad_buff = colourGrad_buff;
	effect.emboss_buff = emboss_buff;
	effect.glass_buff = glass_buff;
	effect.pixel_buff = pixel_buff;

	//addEffect(savedEffect_List, effect);
	return effect;

}
```

```
void ApplySavedEffect(SavedEffect effect) {

        SetCurrentState(effect.EffectInUse);

        m_ScreenBuffer = effect.screen_buff;
        m_ScanlineBuffer = effect.scanline_buff;
        m_VignettingBuffer = effect.vignet_buff;
        m_PosterizeBuffer = effect.posterize_buff;

        m_BloomBuffer = effect.bloom_buff;
        m_Bloom2Buffer = effect.bloom2_buff;
        m_BlurGauBuffer = effect.blurGauBuffer_buff;
        m_ChromaTrueBuffer = effect.chromaTrue_buff;

        m_ChromaStylizedBuffer = effect.chromaStylized_buff;
        m_ColourGradBuffer = effect.colourGrad_buff;
        m_EmbossBuffer = effect.emboss_buff;
        m_GlassBuffer = effect.glass_buff;

        m_PixelBuffer = effect.pixel_buff;

}
```

```
void ResetAll() {

        SetCurrentState(DEFAULT_STATE_);

        m_ScreenBuffer = ScreenBuffer();
        m_ScanlineBuffer = ScanlineBuffer();
        m_VignettingBuffer = VignettingBuffer();
        m_PosterizeBuffer = PosterizeBuffer();

        m_BloomBuffer = BloomBuffer();
        m_Bloom2Buffer = Bloom2Buffer();
        m_BlurGauBuffer = BlurGauBuffer();
        m_ChromaTrueBuffer = ChromaTrueBuffer();

        m_ChromaStylizedBuffer = ChromaStylizedBuffer();
        m_ColourGradBuffer = ColourGradBuffer();
        m_EmbossBuffer = EmbossBuffer();
        m_GlassBuffer = GlassBuffer();

        m_PixelBuffer = PixelBuffer();

}
```

```
// - current pipeline
void SetCurrentState(ShaderToggles effect = DEFAULT_STATE_)
{
        CURRENT_STATE_ = effect;
        m_Pipeline_dirty = true;
}
```

```
SKTBD_PRIMITIVE_TOPOLOGY Topology = SKTBD_PRIMITIVE_TOPOLOGY_TRIANGLELIST;

virtual void Init() override;

virtual void Init(SamplerDesc Sampler)
{
	StaticSamplerDesc = Sampler;
	Init();
}

virtual void Init(size_t TriangleBuffersizeOverride, size_t InstanceBufferSizeOverride)
{
	DynamicBufferSize = ROUND_UP(TriangleBuffersizeOverride, GraphicsConstants::DEFAULT_RESOURCE_ALIGNMENT);
	InstanceDataBufferSize = ROUND_UP(InstanceBufferSizeOverride, GraphicsConstants::DEFAULT_RESOURCE_ALIGNMENT);
	Init();
}
virtual void Init(size_t TriangleBufferSizeOverride, size_t InstanceBufferSizeOverride, SamplerDesc Sampler)
{
	DynamicBufferSize = ROUND_UP(TriangleBufferSizeOverride, GraphicsConstants::DEFAULT_RESOURCE_ALIGNMENT);
	InstanceDataBufferSize = ROUND_UP(InstanceBufferSizeOverride, GraphicsConstants::DEFAULT_RESOURCE_ALIGNMENT);

	StaticSamplerDesc = Sampler;
	Init();
}
virtual void Init(size_t TriangleBufferSizeOverride, size_t InstanceBufferSizeOverride, SamplerDesc Sampler, size_t pennyBufferSizeOverRide)
	//probably not nessasary
{
	DynamicBufferSize = ROUND_UP(TriangleBufferSizeOverride, GraphicsConstants::DEFAULT_RESOURCE_ALIGNMENT);
	InstanceDataBufferSize = ROUND_UP(InstanceBufferSizeOverride, GraphicsConstants::DEFAULT_RESOURCE_ALIGNMENT);
	pennyBufferSize = ROUND_UP(pennyBufferSizeOverRide, GraphicsConstants::DEFAULT_RESOURCE_ALIGNMENT);

	StaticSamplerDesc = Sampler;
	Init();
}
```

```
void SetLights(std::vector<Light> Lights) { m_Lights = Lights; };

void AddLight(Light light) { m_Lights.push_back(light); }
Light& GetLight(uint32_t idx) { return m_Lights[idx]; }

//RendererConfiguration
ShaderInputLayoutRef GetRootSig() { return RootSig; };
ShaderToggles GetShaderToggles() const { return CURRENT_STATE_; };

//Debug normals
void SetDrawDebugNormals(bool DrawNormals) { m_VisualiseNormals = DrawNormals; }

void SetAmbientLight(float3 light_colour) {
	m_Frame.AmbientLight = light_colour;
}
```

The Shader library

Structs

Quick sentence about the structs.

```
#ifndef STRUCTS_Penny2
#define STRUCTS_Penny2

//Para == parameters
```

```
struct ScreenPara //not working
{
    float2 screenSize_;
    int PushData_instanceNo;
    int padding;
};
```

```
struct scanlinePara
{
    float2 screenSize_;
    float LineThickness_;
    float DimmedFactor_;

    float transferPower_;
    float vertical_;
    int PushData_instanceNo;
    float padding;
};
```

Parameter	Description
LineThickness_	Thickness of each scanline or grid cell
DimmedFactor_	Multiplier to darken non-highlighted pixels
transferPower_	How much to brighten highlight regions
vertical_	Controls orientation: horizontal/vertical/grid
screenSize_	Screen resolution (used to convert UVs)

```
struct vignettingPara //
{

    float2 screenSize_;
    float2 padding;

    float innerRadius;
    float outerRadius;
    float opacity;
    int PushData_instanceNo;
};
```

```
  struct posterizePara //
  {
      float3 step_areas;
      uint PushData_instanceNo;
  };
```

```
struct bloomPara //
{
    int PushData_instanceNo;
    int width;
    float angleSteps;
    float radiusSteps;

    float ampFactor;
    float3 padding; // align to 16 bytes
};
```

```
 struct bloom2Para  //
 {
     int PushData_instanceNo;
     float ampFactor;
     float threshold;
     float padding0;
 };
```

```
struct blurGauPara  //
{
   float Luminance;
   float2 screenSize_;
   float padding0;

   float weight[8]; //last values ar padding

   float texscrMultiplier[8];//last values ar padding

   //float weight[5];
   //float padding1[3]; // padding to align next array

   //float texscrMultiplier[5];
   //float padding2[3]; // padding for alignment
};
```

```
struct ChromaTruePara  //
{
    int PushData_instanceNo;
    float2 screenSize_;
    float aberrationFactor;

    float aberrationFactor_x;
    float aberrationFactor_y;
    float2 padding; // padding for alignment
};
```

```
struct ChromaStylizedPara //might nc
                          //not wc
{
    int PushData_instanceNo;
    float3 padding0;

    float4 chromaticWeights1;
    float4 chromaticWeights2;
    float4 chromaticOffset1;
    float4 chromaticOffset2;
};
```

```
struct colourGradPara
{
    int PushData_instanceNo;
    float3 padding0;

    float3 startColor;
    float padding1;

    float3 endColor;
    float padding2;
};
```

```
struct EmbossPara  //
{
    int PushData_instanceNo;
    float2 screenSize_;
    float padding0;

    int greyScale;
    float width_;
    float height_;
    float padding1;
};
```

```
struct glassPara  //
{
    int PushData_instanceNo;
    float2 screenSize_;
    float2 panelSize;

    float2 padding; // for alignment
};
```

```
struct pixelPara  //
{
    int PushData_instanceNo;
    float2 screenSize_;
    float2 pixelSize;

    float2 padding; // for alignment
};

#endif
```

Shaders

Quick sentence about the library.

Scanline Pixel Shader

```
#include "../../CMP203/CMP203_Shaders/Structs.hlsli" #include "PennyStructs_2.hlsli"

//sampler registers SamplerState g_sampler : register(s0); //buffers ConstantBuffer PushData :
register(b0, space0); StructuredBuffer Instances : register(t0, space0); ConstantBuffer Parameters :
register(b2, space0);

float4 main(VertexOutput input) : SV_Target0 { //get values from buffer float LineThickness =
Parameters.LineThickness_; float DimmedFactor = Parameters.DimmedFactor_;; float transferPower =
Parameters.transferPower_; float vertical = Parameters.vertical_; float2 screenSize =
Parameters.screenSize_;

float4 resultColor;


 // Access the texture with the instance data
Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];
//get pixel colour from sampling texture
float4 textureColor = texture0.Sample(g_sampler, input.uv);

// Convert UVs to actual pixel positions
int pixelX = input.uv.x * screenSize.x;
int pixelY = input.uv.y * screenSize.y;



// Determine if we're in a "band" area or a "gap" based on pixel location
int xType = (pixelX / LineThickness) % 2;
int yType = (pixelY / LineThickness) % 2;

if (vertical != 0.5)//lines
{
    if ((vertical && xType == 0) || (!vertical && yType == 0))
    {
        // Brighten the pixel (emphasize the line)
        resultColor = textureColor + (textureColor * (transferPower * (1 - DimmedFactor)));
    }
    else
    {
        // Dim the pixel
        resultColor = textureColor * DimmedFactor;
    }
```

```
}
else if (vertical == 0.5) // crosshatch / matrix
{
    if (xType == 0 && yType == 0)
    {   // Brighten intersections more
        resultColor = textureColor + (textureColor * 3 * (transferPower * (1 - DimmedFactor)));
    }
    else
    {
        // Dims the left pixels
        resultColor = textureColor * DimmedFactor;
    }
}

return resultColor;


}
```

Parameter	Description
LineThickness_	Thickness of each scanline or grid cell
DimmedFactor_	Multiplier to darken non-highlighted pixels
transferPower_	How much to brighten highlight regions
vertical_	Controls orientation: horizontal/vertical/grid
screenSize_	Screen resolution (used to convert UVs)

Pixel Shader Overview;

This HLSL pixel shader applies a **scanline or crosshatch visual effect** over a textured object. The effect is controlled by a set of adjustable parameters such as line thickness, brightness adjustment, and orientation (vertical, horizontal, or crosshatch).

Shader Inputs and Buffers
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"

These headers define the structures for:

- Vertex/pixel input/output
- Shader constants

SamplerState g_sampler : register(s0);
ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);
ConstantBuffer<scanlinePara> Parameters : register(b2, space0);

Resources:

- g_sampler: Sampler used for texture sampling
- PushData: Holds the current instance index (instanceNo)

- Instances: Buffer holding per-instance texture indices
- Parameters: Holds scanline effect parameters (see above)

Main Breakdown

float4 main(VertexOutput input) : SV_Target0

The pixel shader takes interpolated vertex output and returns a final float4 colour.

---

1. Set Parameters

```
float LineThickness = Parameters.LineThickness_;
float DimmedFactor = Parameters.DimmedFactor_;
float transferPower = Parameters.transferPower_;
float vertical = Parameters.vertical_;
float2 screenSize = Parameters.screenSize_;
```

These are read from the constant buffer to control visual behaviour.

---

2. Sample the Texture

```
Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];
float4 textureColor = texture0.Sample(g_sampler, input.uv);
```

- Retrieves the texture for the current instance using the instanceNo index.
- Samples the texture colour at the UV coordinate.

---

3. Convert UV to Pixel Space

```
int pixelX = input.uv.x * screenSize.x;
int pixelY = input.uv.y * screenSize.y;
```

- Converts normalized UVs into absolute screen pixel coordinates.
- Used to compute which "band" or "gap" a pixel falls into.

---

4. Determine Line Position

```
int xType = (pixelX / LineThickness) % 2;
int yType = (pixelY / LineThickness) % 2;
```

- Calculates whether a pixel is inside a highlighted or dimmed region.
    - Even (0) = line area
    - Odd (1) = gap area

---

5. Apply Effects Based on Orientation

a. Vertical or Horizontal Lines

```
if (vertical != 0.5)
{
   if ((vertical && xType == 0) || (!vertical && yType == 0))
   {
      resultColor = textureColor + (textureColor * (transferPower * (1 - DimmedFactor)));
   }
   else
   {
      resultColor = textureColor * DimmedFactor;
   }
}
```

- If vertical == 1.0, it applies **vertical lines**
- If vertical == 0.0, it applies **horizontal lines**
- Brightens pixels inside line bands; dims others

b. Crosshatch Mode

```
else if (vertical == 0.5)
{
   if (xType == 0 && yType == 0)
   {
      resultColor = textureColor + (textureColor * 3 * (transferPower * (1 - DimmedFactor)));
   }
   else
   {
      resultColor = textureColor * DimmedFactor;
   }
}
```

- When vertical == 0.5, **both x and y line patterns** are applied
- Pixels at intersections (where both xType and yType are 0) are **brightened more**
- All others are dimmed

Output

```
return resultColor;
```

- Returns the final shaded color with the scanline or crosshatch effect applied.

Summary

This pixel shader:

- Samples a texture per instance
- Applies scanline or grid-style overlays
- Uses screen-space math for patterns
- Allows dynamic control of line thickness, dimming, and brightness through constants

Vignetting Pixel Shader

```
#include "../../CMP203/CMP203_Shaders/Structs.hlsli" #include "PennyStructs_2.hlsli"

//sampler registers SamplerState g_sampler : register(s0); #define PI acos(-1)

//buffers ConstantBuffer PushData : register(b0, space0); StructuredBuffer Instances : register(t0,
space0);

ConstantBuffer Parameters : register(b2, space0); //also not working

float4 main(VertexOutput input) : SV_Target0 { // assign parameters from constant buffer float innerRadius
= Parameters.innerRadius; float outerRadius = Parameters.outerRadius; float opacity = Parameters.opacity;

// Access the texture with the instance data
Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];
//gets pixel colour from sampling texture
float4 textureColor = texture0.Sample(g_sampler, input.uv);

// Adds vertical dimming effect
float verticalDim = 0.5 + sin(input.uv.y * PI) * 0.9;

// Convert UVs from [0,1] to [-1,1] coordinate space for radius calculation
float xTrans = (input.uv.x * 2) - 1;
float yTrans = 1 - (input.uv.y * 2);

// calculates the radial distance from the center of the screen
float radius = sqrt(pow(xTrans, 2) + pow(yTrans, 2));

// Calculate how far past the innerRadius the pixel is
float subtraction = max(0, radius - innerRadius) / (outerRadius - innerRadius);
float factor = 1 - subtraction;

// Apply the vignette effect based on the calculated factor
float4 vignetColor = textureColor * factor;

// Apply additional vertical fading
vignetColor *= verticalDim;



// Blend with original coloure based on opacity
vignetColor *= opacity;
textureColor *= 1 - opacity;



// Combine both to get the final colour
return textureColor + vignetColor;


}
```

Posterize Pixel Shader

```
#include "../../CMP203/CMP203_Shaders/Structs.hlsli" //#include "PennyStructs.hlsli" #include
"PennyStructs_2.hlsli"
```

```
// sampler registers SamplerState g_sampler : register(s0); #define PI acos(-1)

ConstantBuffer PushData : register(b0, space0);

StructuredBuffer Instances : register(t0, space0); ConstantBuffer Parameters : register(b2, space0); //

float4 main(VertexOutput input) : SV_Target0 { // Access the texture with the instance data Texture2D
texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];

//sample to get pixle colour
float4 textureColour = texture0.Sample(g_sampler, input.uv);

//Parameters controlling how many steps each channel is broken into -
// e.g., (4, 4, 4) reduces each channel to 4 values/colours
float3 step_areas = Parameters.step_areas;

// Scale the colours by number of steps
textureColour.r = textureColour.r * step_areas.r;
textureColour.g = textureColour.g * step_areas.g;
textureColour.b = textureColour.b * step_areas.b;

 // round down the values to lock them into steps, make sure the colour is opaque by having the last value
be 1
float4 result = float4((floor(textureColour.r) + 0.5) / step_areas.r, (floor(textureColour.g) + 0.5) /
step_areas.g, (floor(textureColour.b) + 0.5) / step_areas.b, 1);

return result;


}
```

bloom Pixel Shader

```
#include "../../CMP203/CMP203_Shaders/Structs.hlsli" #include "PennyStructs_2.hlsli"

// Texture and sampler registers SamplerState g_sampler : register(s0); #define PI acos(-1)

//buffers ConstantBuffer PushData : register(b0, space0); StructuredBuffer Instances : register(t0,
space0); ConstantBuffer bloomParaBuffer : register(b2, space0);

float4 main(VertexOutput input) : SV_Target0

{ // Access the texture with the instance data Texture2D texture0 =
ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];

// assign bloom parameters from constant buffer - this was done for time and sake of ease (change layter)
int width = bloomParaBuffer.width; //10
float angleSteps = bloomParaBuffer.angleSteps; //12
float radiusSteps = bloomParaBuffer.radiusSteps;
float ampFactor = bloomParaBuffer.ampFactor; //

// Sample colour of the current pixel
float4 c0 = texture0.Sample(g_sampler, input.uv);
float4 origColour_ = c0;
float4 accumulatedColour_ = { 0, 0, 0, 0 };

// Set min and max radius for sampling
double minRadius_ = (0.0 / width);
double maxRadius_ = (10.0 / width);

// Calculate total steps and increments
```

```
int totalSteps_ = radiusSteps * angleSteps;
float angleDelta_ = (2 * PI) / angleSteps;
float radiusDelta_ = (maxRadius_ - minRadius_) / radiusSteps;

// Loop over the radius and angles to get surrounding pixel colours
for (int radiusStep_ = 0; radiusStep_ < radiusSteps; radiusStep_++)
{
    float radius = minRadius_ + radiusStep_ * radiusDelta_;

    for (float angle = 0; angle < (2 * PI); angle += angleDelta_)
    {
        float2 currentCoord;

         // Compute offset from current pixel
        float xDiff = radius * cos(angle);
        float yDiff = radius * sin(angle);

        currentCoord = input.uv + float2(xDiff, yDiff);

        // Sample the surrounding pixel colour
        float4 currentColour = texture0.Sample(g_sampler, currentCoord);

        // Weight closer samples more than distant ones
        float currentFraction = ((float) (radiusSteps + 1 - radiusStep_)) / (radiusSteps + 1);

        // Accumulate weighted color
        accumulatedColour_ += currentFraction * currentColour / totalSteps_;

    }
}

// Final output: base color + bloom/glow effect
float4 outputPixel = texture0.Sample(g_sampler, input.uv);
outputPixel += accumulatedColour_ * ampFactor;

return outputPixel;


}
```

Bloom 2 Pixel Shader

```hlsl
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"

// Texture and sampler registers
SamplerState g_sampler : register(s0);
#define PI acos(-1)

//buffers
ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);
ConstantBuffer<bloom2Para> bloom2ParaBuffer : register(b2, space0);

float4 main(VertexOutput input) : SV_Target0
{
    // Access the texture with the instance data
    Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];


    // assign bloom parameters from constant buffer
    float ampFactor = bloom2ParaBuffer.ampFactor; //
    float threshold = bloom2ParaBuffer.threshold; //

    float4 bloomColour = texture0.Sample(g_sampler, input.uv); //bloom colour

    // if pixel colour is lower than or == threshhold
    if ((bloomColour.x <= threshold) || (bloomColour.y <= threshold) || (bloomColour.z <= threshold))
    {
        bloomColour = float4(0.0f, 0.0f, 0.0f, 0.0f);//set current pixle colour to 0
    }

    // Sample orignal pixle colour
    float4 outputPixel = texture0.Sample(g_sampler, input.uv);

    //add bloom (* by amp factor to ajust brightness) to the orignal texture/pixel colour
    outputPixel += (bloomColour * ampFactor);

    //return the new colour
    return outputPixel;

}
```

Blur Gau Para

```hlsl
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"

// sampler register
SamplerState g_sampler : register(s0);
#define PI acos(-1)

//buffers
ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);
ConstantBuffer<blurGauPara> blurGauParaBuffer : register(b2, space0);

float4 main(VertexOutput input) : SV_Target0
{
    //buffer values
    float screenWidth = 1366;
    float screenHeight = 768;

    float4 colour;

    Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];
    float4 textureColor = texture0.Sample(g_sampler, input.uv);

        // Create the weights that each neighbor pixel will contribute to the blur.
    float weight[5];// = blurGauParaBuffer.weight;
    float texscrMultiplier[5];// = blurGauParaBuffer.texscrMultiplier;

    for (int i = 0; i < 5; i++)
    {
       //    weight[i] = 0.2;
        texscrMultiplier[i] = i;
        weight[i] = blurGauParaBuffer.weight[i];
       // texscrMultiplier[i] = blurGauParaBuffer.texscrMultiplier[i];
    }

    //--
    float texelSizeWidth = 1.0f / screenWidth;
    float texelSizeHeight = 1.0f / screenHeight;

    // Initialize the colour to black.
    colour = float4(0.0f, 0.0f, 0.0f, 0.0f);

    // Add the pixels to the colour by the specific weight of each.
    colour += texture0.Sample(g_sampler, input.uv + float2(texelSizeWidth * -texscrMultiplier[4], screenHeight * -texscrMultiplier[4])) * weight[4];
    colour += texture0.Sample(g_sampler, input.uv + float2(texelSizeWidth * -texscrMultiplier[3], screenHeight * -texscrMultiplier[3])) * weight[3];
    colour += texture0.Sample(g_sampler, input.uv + float2(texelSizeWidth * -texscrMultiplier[2], screenHeight * -texscrMultiplier[2])) * weight[2];
    colour += texture0.Sample(g_sampler, input.uv + float2(texelSizeWidth * -texscrMultiplier[1], screenHeight * -texscrMultiplier[1])) * weight[1];
    colour += texture0.Sample(g_sampler, input.uv) * weight[0];
    colour += texture0.Sample(g_sampler, input.uv + float2(texelSizeWidth * texscrMultiplier[1], screenHeight * texscrMultiplier[1])) * weight[1];
    colour += texture0.Sample(g_sampler, input.uv + float2(texelSizeWidth * texscrMultiplier[2], screenHeight * texscrMultiplier[2])) * weight[2];
    colour += texture0.Sample(g_sampler, input.uv + float2(texelSizeWidth * texscrMultiplier[3], screenHeight * texscrMultiplier[3])) * weight[3];
    colour += texture0.Sample(g_sampler, input.uv + float2(texelSizeWidth * texscrMultiplier[4], screenHeight * texscrMultiplier[4])) * weight[4];

    colour = colour * blurGauParaBuffer.Luminance; // correcting Luminance

        // Set the alpha channel to one.
    colour.a = 1.0f;

    return colour;

}
```

Chroma True Pixel Shader

```hlsl
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"

//sampler registers
SamplerState g_sampler : register(s0);
#define PI acos(-1)


//buffers
ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);
ConstantBuffer<ChromaTruePara> ChromaTrueParaBuffer : register(b2, space0);


float4 main(VertexOutput input) : SV_Target0
{
  // Access the texture with the instance data
    Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];

    //buffer values assigned
    float aberrationFactor = ChromaTrueParaBuffer.aberrationFactor;
    float aberrationFactor_x = ChromaTrueParaBuffer.aberrationFactor_x;
    float aberrationFactor_y = ChromaTrueParaBuffer.aberrationFactor_y;

    // calculate distances from the reference point
    float distX = input.uv.x - aberrationFactor_x;
    float distY = -input.uv.y + aberrationFactor_y;

    //init all rgb colour channles first
    float2 rColour, gColour, bColour;

    // Red channel gets pushed outward
    rColour.x = input.uv.x + aberrationFactor * distX;
    rColour.y = input.uv.y + aberrationFactor * distY;

    // Green channel stays centered (acts like an anchor)
    gColour.x = input.uv.x;
    gColour.y = input.uv.y;

    // Blue channel gets pulled inward (opposite to red)
    bColour.x = input.uv.x - aberrationFactor * distX;
    bColour.y = input.uv.y - aberrationFactor * distY;

    // Sample color from each adjusted channel
    float4 rTex = texture0.Sample(g_sampler, rColour);
    float4 gTex = texture0.Sample(g_sampler, gColour);
    float4 bTex = texture0.Sample(g_sampler, bColour);

    // add up final pixel color with shifted RGB values and make sure opacity is 1(full)
    float4 result = { rTex[0], gTex[1], bTex[2], 1 };

    return result;

}
```

Chroma Stylized Pixel Shader

```hlsl
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"


// sampler registers
SamplerState g_sampler : register(s0);
#define PI acos(-1)

//buffers
ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);
ConstantBuffer<ChromaStylizedPara> ChromaStylizedParaBuffer : register(b2, space0);


float4 main(VertexOutput input) : SV_Target0
{

     // Access texture with instance data
    Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];

    //assign the buffer values from constBuffer
    float4 chromaticWeights1 = ChromaStylizedParaBuffer.chromaticWeights1;
    float4 chromaticWeights2 = ChromaStylizedParaBuffer.chromaticWeights2;
    float4 chromaticOffset1 = ChromaStylizedParaBuffer.chromaticOffset1;
    float4 chromaticOffset2 = ChromaStylizedParaBuffer.chromaticOffset2;

    //set all colours initially to 0
    float4 vColor_chorma1 = float4(0, 0, 0, 0);
    float4 vColor_chorma2 = float4(0, 0, 0, 0);
    float4 vColor_chorma = float4(0, 0, 0, 0);


    //1
    for (int i = 0; i < chromaticOffset1.z; i++)
    {
        vColor_chorma1 += chromaticWeights1 * texture0.Sample(g_sampler, input.uv + chromaticOffset1.xy);
    }
    vColor_chorma1 = vColor_chorma1 / chromaticOffset1.z;

    //2
    for (int i = 0; i < chromaticOffset2.z; i++)
    {
        vColor_chorma2 += chromaticWeights2 * texture0.Sample(g_sampler, input.uv + chromaticOffset2.xy);
    }
    vColor_chorma2 = vColor_chorma2 / chromaticOffset2.z;

    //combine all the pixel colours
    vColor_chorma = vColor_chorma2 + vColor_chorma1;

    return vColor_chorma; //NUM_SAMPLES

}
```

Colour Grad Pixel Shader

```hlsl
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"

//sampler registers
SamplerState g_sampler : register(s0);
#define PI acos(-1)

//buffers
ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);

ConstantBuffer<colourGradPara> colourGradParaBuffer : register(b2, space0);

float4 main(VertexOutput input) : SV_Target0
{
    //bufferValues
    float3 startColor = colourGradParaBuffer.startColor;
    float3 endColor = colourGradParaBuffer.endColor;

     // Access the texture with the instance data
    Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];
    //get pixel colour using sampler
    float4 textureColour = texture0.Sample(g_sampler, input.uv);

    //lumanince is calcluate
    float lum = (textureColour.r + textureColour.g + textureColour.b) / 3.0;
    float lumComp = 1.0 - lum;

        // Create colour based on luminance and the colour range
    float4 modColour;
    modColour.r = lum * startColor.r + lumComp * endColor.r;
    modColour.g = lum * startColor.g + lumComp * endColor.g;
    modColour.b = lum * startColor.b + lumComp * endColor.b;
    modColour.a = 1;

        // Multiply the original colour by the modified color
    float4 result = textureColour * modColour;

        //=-- Luminance restoration
    float resultLum = (result.r + result.g + result.b) / 3.0;
    result *= lum / resultLum;

    return result;


}
```

Emboss Pixel Shader

```hlsl
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"


//sampler registers
SamplerState g_sampler : register(s0);
#define PI acos(-1)

//buffers
ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);
ConstantBuffer<EmbossPara> EmbossParaBuffer : register(b2, space0);

float4 main(VertexOutput input) : SV_Target0
{

    //float2 screenSize = (1366, 768);
    float2 screenSize = EmbossParaBuffer.screenSize_;

    bool greyScale = EmbossParaBuffer.greyScale;//some times this could potentially produce a really coo

     // Access the texture with the instance data
    Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];

    // assign values from constant buffer
    float width_ = EmbossParaBuffer.width_;
    float height_ = EmbossParaBuffer.height_;

    // Calculate one-pixel step sizes in UV space
    float dx = 1 / width_;
    float dy = 1 / height_;

    // Sample neighboring pixels in a cross-diagonal pattern
    float4 textureColour_0 = texture0.Sample(g_sampler, input.uv + float2(-dx, -dy));
    float4 textureColour_1 = texture0.Sample(g_sampler, input.uv + float2(0, -dy));
    float4 textureColour_2 = texture0.Sample(g_sampler, input.uv + float2(-dx, 0));
    float4 textureColour_3 = texture0.Sample(g_sampler, input.uv + float2(dx, 0));
    float4 textureColour_4 = texture0.Sample(g_sampler, input.uv + float2(0, dy));
    float4 textureColour_5 = texture0.Sample(g_sampler, input.uv + float2(dx, dy));

    // Combine sampled values to emphasize edges and depth (emboss calculation)
    float4 textureColor = (-textureColour_0 - textureColour_1 - textureColour_2 + textureColour_3 + text

    //If grayscale is enabled, convert the result to greyscale
    if (greyScale == true)
    { // Average the RGB channels and add 0.5 to adjust the brightness
       textureColor = (textureColor.r + textureColor.g + textureColor.b) / 3 + 0.5; // grey scale
    }

    return textureColor;

}
```

glass Pixel Shader

```hlsl
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"


//sampler register
SamplerState g_sampler : register(s0);
#define PI acos(-1)


//buffers
ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);

ConstantBuffer<glassPara> glassParaBuffer : register(b2, space0);

float4 main(VertexOutput input) : SV_Target0
{
    //bufferValues
    glassParaBuffer.screenSize_;

    float width = 1366;//screen size
    float height = 768;//screen size

    // buffer values assigned
    // -- Panel size controls frequency of the distortion pattern
    float xSize = glassParaBuffer.panelSize.x;
    float ySize = glassParaBuffer.panelSize.y;

    // Access the texture with the instance data
    Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];

    float2 tex2;

    // Determines position within the panel grid
    int xSubpixel = (input.uv.x * width) % xSize;
    int ySubpixel = (input.uv.y * height) % ySize;

    //calculates a small offset based on position within the panel grid
    float2 offsets = { 0.8 * xSubpixel / width, 0.8 * ySubpixel / height };

     // Apply the offset to the input UVs to simulate refraction through glass
    input.uv += offsets;

    // additional offset for sampling
    int offset = 2;
    tex2.x = input.uv.x + (offset / width);
    tex2.y = input.uv.y + (offset / height);

    //sample colur with offset
    float4 textureColour_1 = texture0.Sample(g_sampler, tex2);

    return textureColour_1;


}
```

Pixelate Pixel Shader

```hlsl
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"

// Texture and sampler registers
SamplerState g_sampler : register(s0);
#define PI acos(-1)

//buffers
ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);

ConstantBuffer<pixelPara> pixelParaBuffer : register(b2, space0); //


float4 main(VertexOutput input) : SV_Target0
{
   // Size of each virtual "pixel"
    float2 pixelSize = pixelParaBuffer.pixelSize;

    float2 tex1;

    // assigning pixel size with values from buffers
    float width_ = pixelSize.x;
    float height_ = pixelSize.y;

    // Convert UV coordinates to screen-space "pixel" positions
    int pixelX = input.uv.x * width_;
    int pixelY = input.uv.y * height_;

    //Align UVs to the top-left corner of the current pixel block
    tex1.x = ((pixelX / pixelSize) * pixelSize) / width_;
    tex1.y = ((pixelY / pixelSize) * pixelSize) / height_;

     // Access the texture with the instance data
    Texture2D texture0 = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];

    // Use the UVs to get a "blocky" pixelated color
    float4 textureColor = texture0.Sample(g_sampler, tex1);

    return textureColor;

}
```

Default shader

```
#include "../../CMP203/CMP203_Shaders/Structs.hlsli"
#include "PennyStructs_2.hlsli"

// Texture and sampler registers
SamplerState g_sampler : register(s0);
#define PI acos(-1)

ConstantBuffer<Constants> PushData : register(b0, space0);
StructuredBuffer<InstanceData> Instances : register(t0, space0);

cbuffer FRAMEDATA : register(b1, space0)
{
    Frame FrameData;
}

ConstantBuffer<ScreenPara> para : register(b2, space0);


float4 main(VertexOutput input) : SV_Target0
{
    //this just passes through the texture colour with no changes - basically regular texture pixle shader

    Texture2D texture = ResourceDescriptorHeap[Instances[PushData.instanceNo].textureIDX];

    float4 c0 = texture.Sample(g_sampler, input.uv);

    return c0;

}
```

Vertex shader

This was originally from the 203 extenders