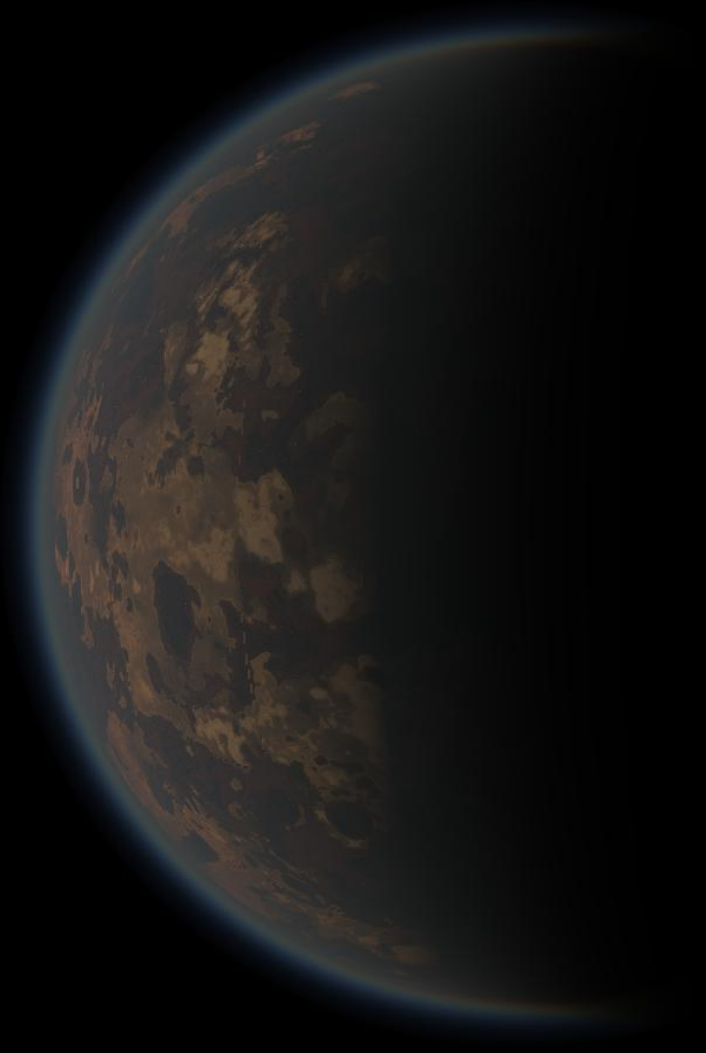
CyberEngine



MANUEL MARQUEZ

manuelxmarquez@gmail.com
manuelmarquez.ca

Contents

Purpose

The primary purpose of this document is to highlight my experience with video game engines and graphics programming. This work, created entirely by myself, demonstrates the ability to design and implement a complete real-time rendering engine, integrating low-level graphics APIs and engine subsystems for high-performance visual applications.

Overview

CyberEngine is a fully custom game engine written in C#. It features a dedicated Visual Studio project type for game content compilation and packaging with NuGet support, user interface framework, and physics engine. The rendering system supports multiple graphics APIs, including Direct3D 11, Direct3D 12, Vulkan, and OpenGL, with both rasterization and ray tracing capabilities.

While primarily targeting Windows, most engine systems are also supported on Linux. All rendering backends are abstracted behind a unified interface, allowing a single application to target multiple APIs. Since not all APIs provide identical capabilities, the engine is designed around a lowest common feature set, following Direct3D's architecture and terminology as the primary reference.

Graphics

The engine currently leverages Vortice, an open-source project that uses SharpGen to automatically generate C# bindings for native graphics libraries. The wrapper enables low-level access to modern rendering APIs directly from managed code. I have also contributed several bug fixes and feature updates to Vortice to improve its reliability and integration with CyberEngine.

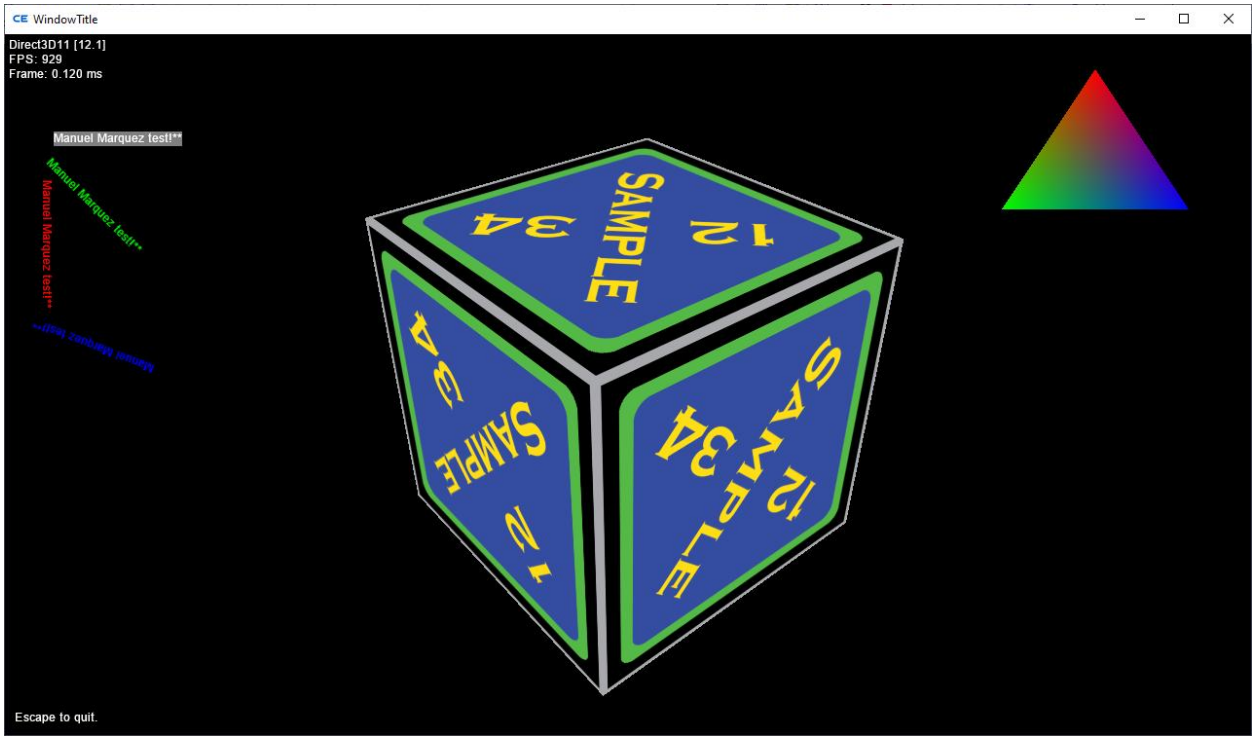


Figure 1 Test application for 2D, 3D, and sprites using Direct3D 11.

Figure 2 Test application using Direct3D 12.

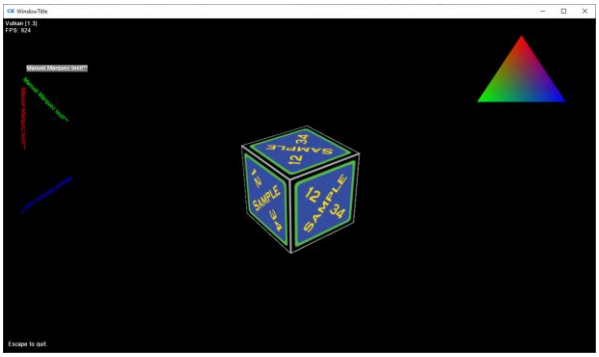


Figure 3 Test application using Vulkan.

OpenGL

For OpenGL, a code generator parses the official XML specification to automatically produce the C# bindings. After the OpenGL context is created, the engine initializes all function pointers by iterating through the generated methods and resolving each one via its proc address.

Below is an example of the method `glDrawArrays`.

```
<Function Name="glDrawArrays">
  <Parameter Name="mode">
    <Type Name="PrimitiveType" />
  </Parameter>
  <Parameter Name="first">
    <Type Name="GLint" />
  </Parameter>
  <Parameter Name="count">
    <Type Name="GLsizei" />
  </Parameter>
</Function>
```

Figure 4 Snippet from the OpenGL specification for the method `glDrawArrays`.

```
[Feature("GL_VERSION_1_1")]
public delegate void DrawArraysDelegate(Enumerators.PrimitiveType mode, int first,
                                        uint count);
```

Figure 5 Snippet of the auto generated C# delegate for glDrawArrays.

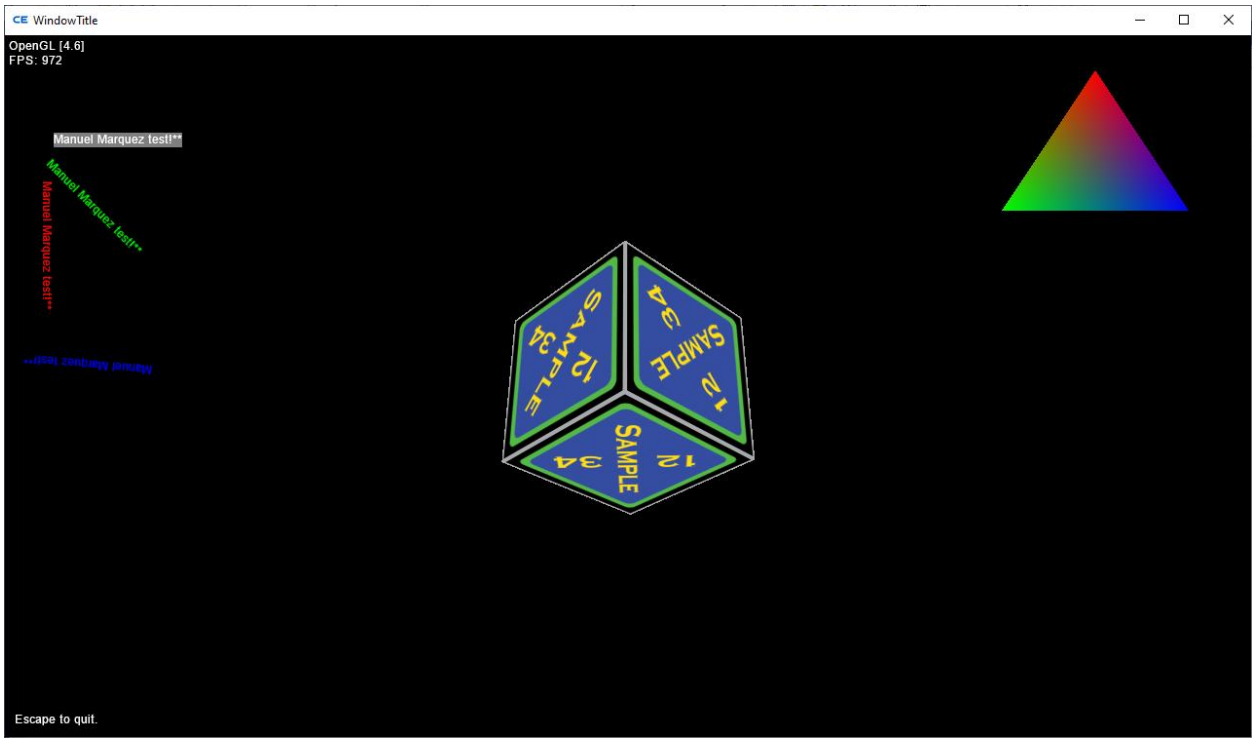


Figure 6 Test application using OpenGL.

PHYSICALLY BASED RENDERING (PBR)

CyberEngine supports a Physically Based Rendering (PBR) pipeline. Auto exposure is calculated by creating a luminance histogram in a compute shader and processing the results on the CPU. Some images from a test program are shown below.

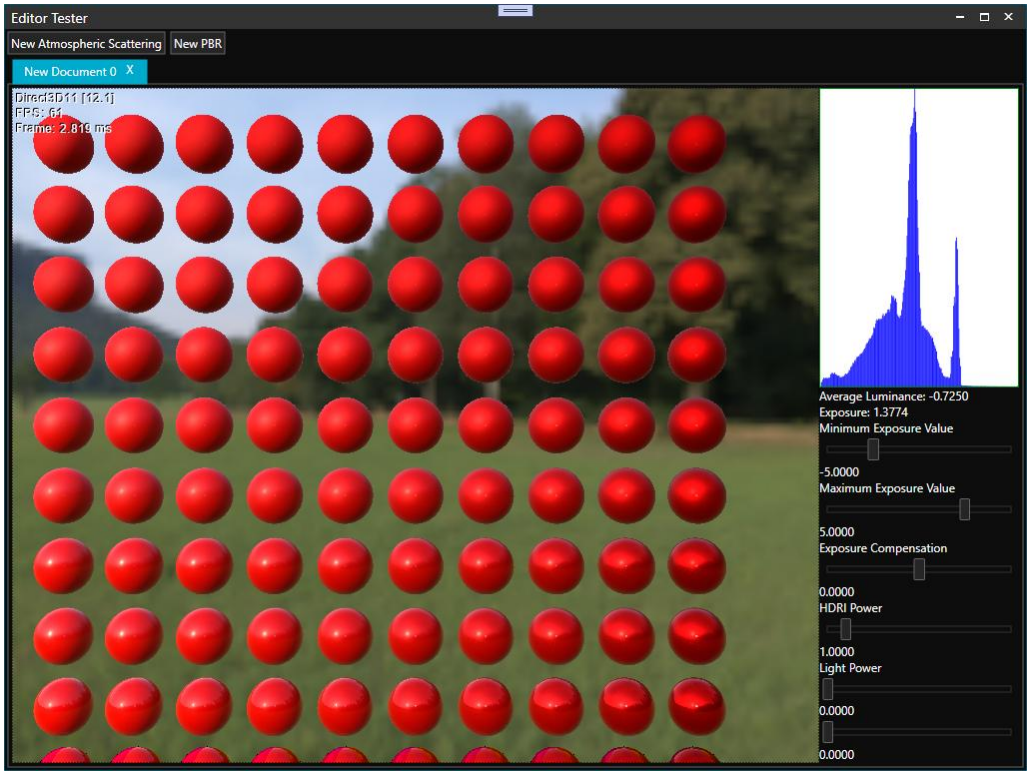


Figure 7 PBR rendering example in a WPF application. Top right illustrating a luminance histogram computed on the GPU. Additional controls allow for tuning the exposure and intensity of the lights and environment map. The spheres vary in metalness/dielectric across the x-axis and roughness along the y-axis.

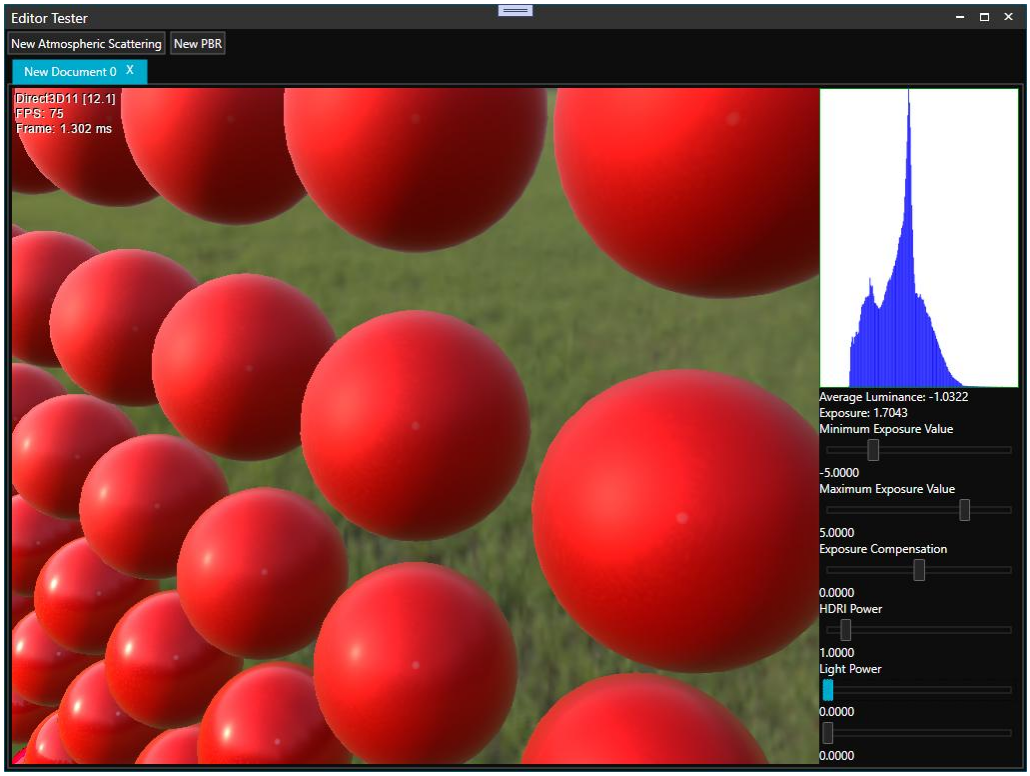


Figure 8 Another view of the PBR spheres with low roughness near the bottom.

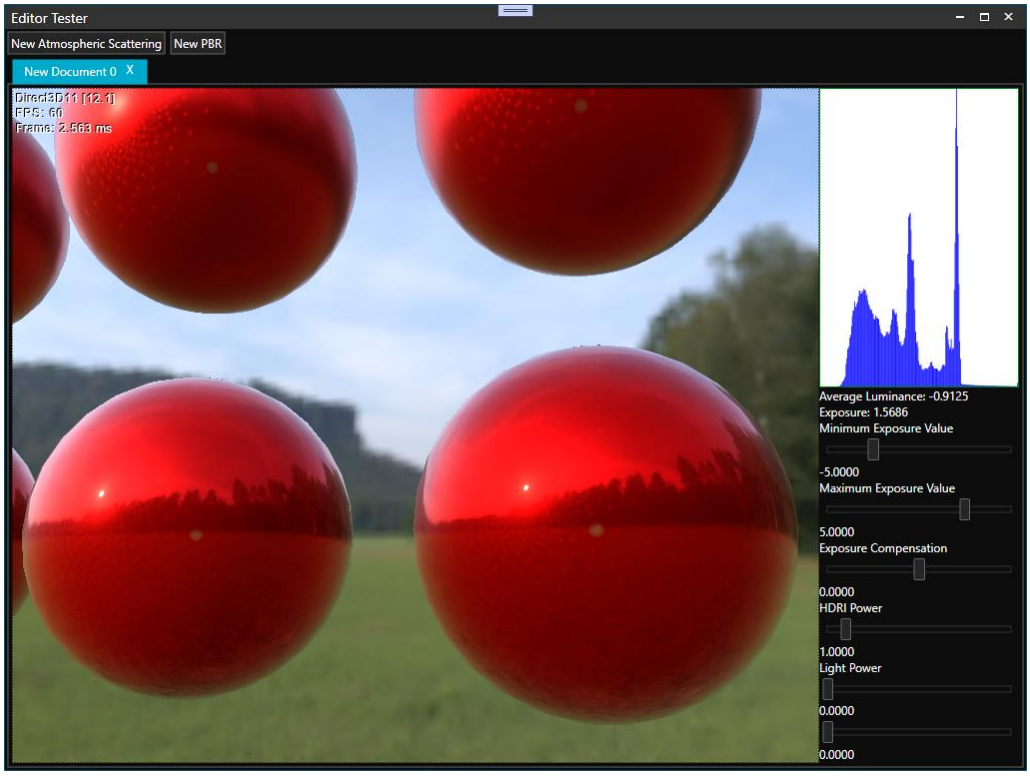


Figure 9 Close up image of a low roughness metallic sphere.

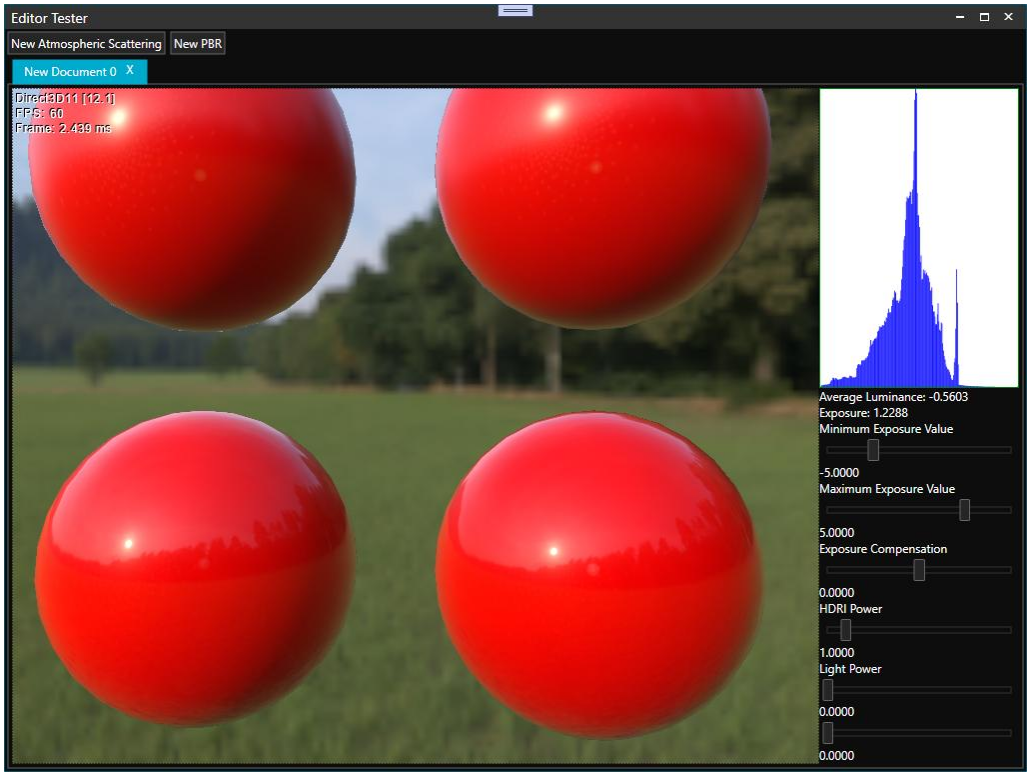


Figure 10 Close up image of a low roughness dielectric (plastic) sphere.

ATMOSPHERIC SCATTERING

A current project requires an atmospheric scattering shader to render planets. A few images from an editor and different settings are shown below.

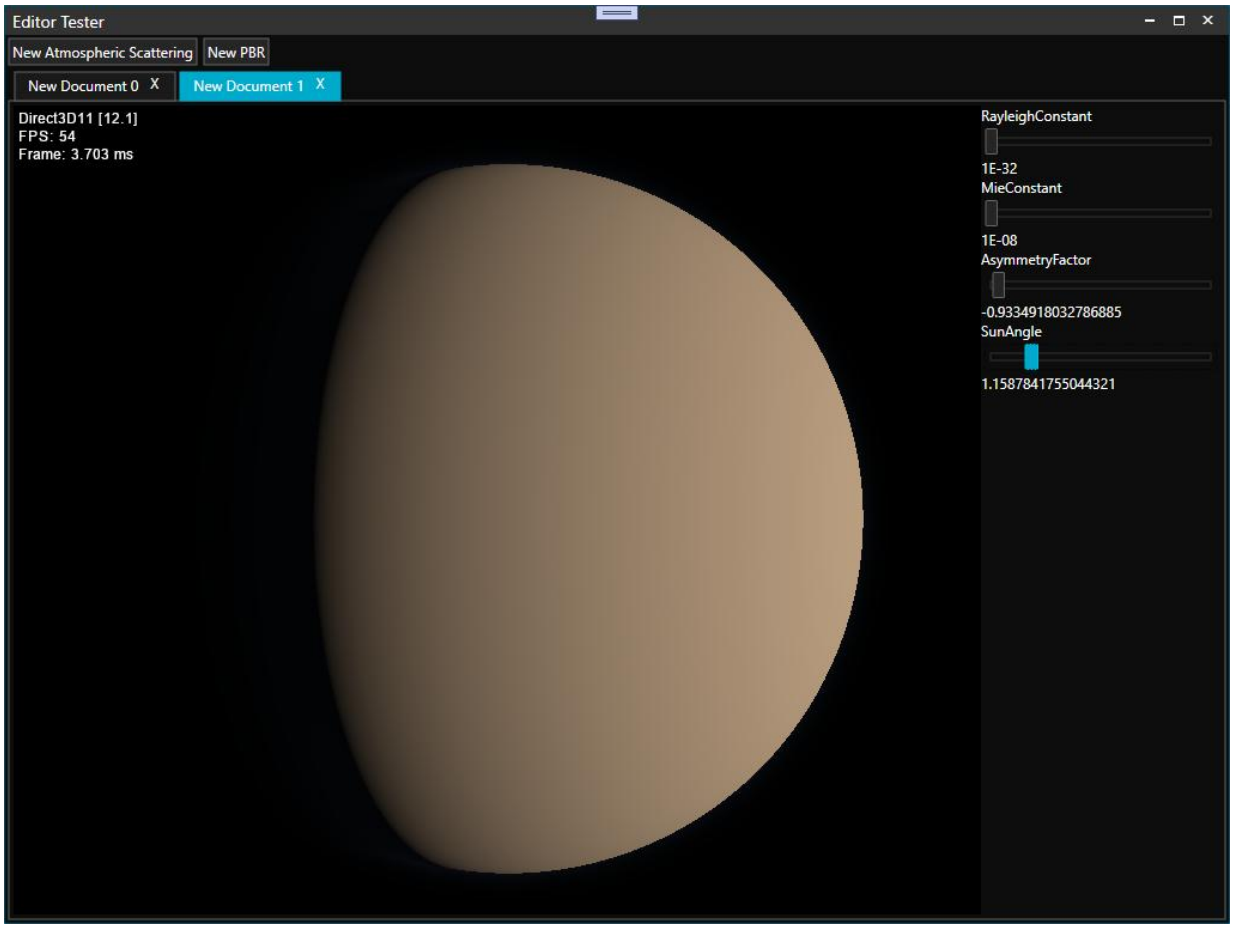


Figure 11 Atmospheric scattering example with a no Rayleigh or Mie constant.

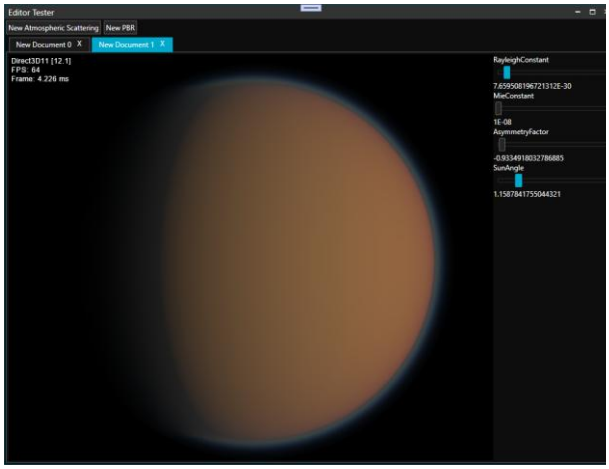


Figure 12 Atmospheric scattering example with low Rayleigh constant. Note the blue light being scattering within the atmosphere.

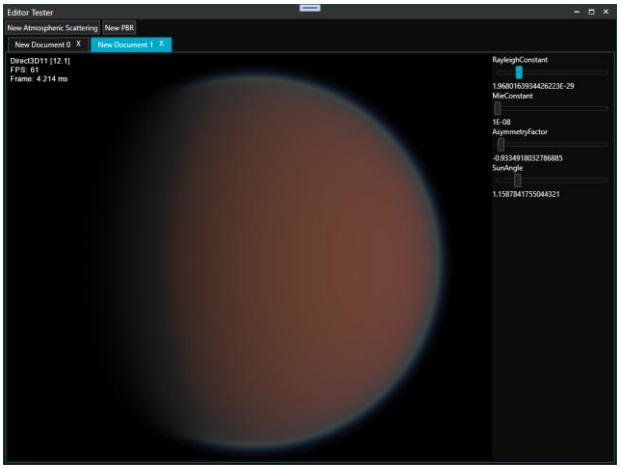


Figure 13 Atmospheric scattering example with medium Rayleigh constant.

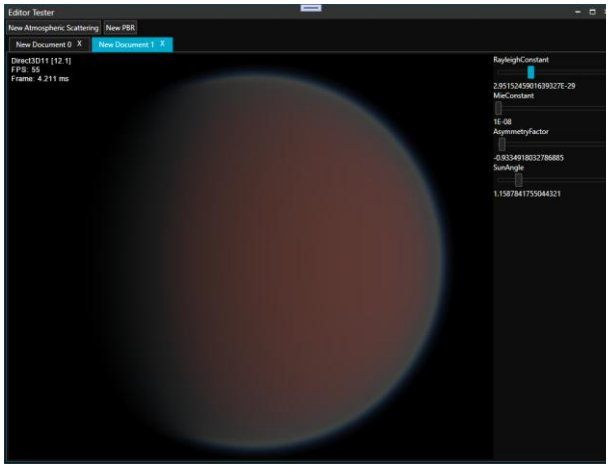


Figure 14 Atmospheric scattering example with high Rayleigh constant. Note the red shift as the blue light is scattered away.

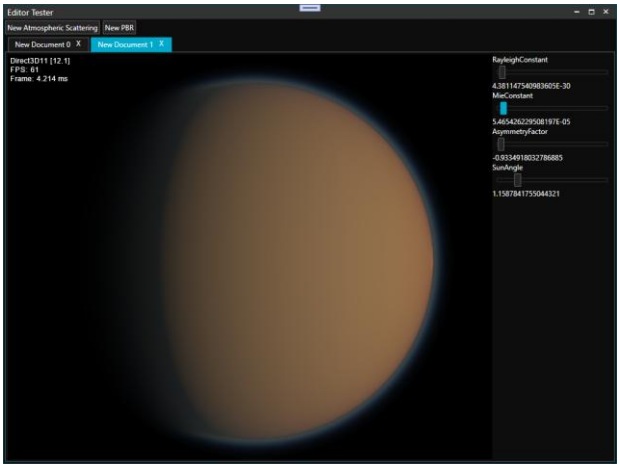


Figure 15 Atmospheric scattering example with low Mie constant. Note the cloudy appearance as Mie models the scattering of aerosols which scatterings all wavelengths equally.

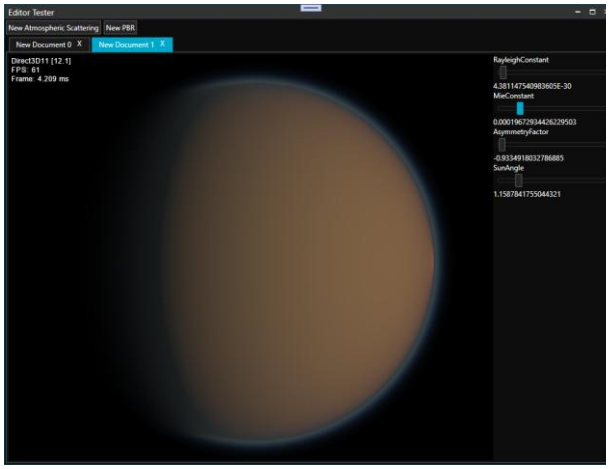


Figure 16 Atmospheric scattering example with medium Mie constant.

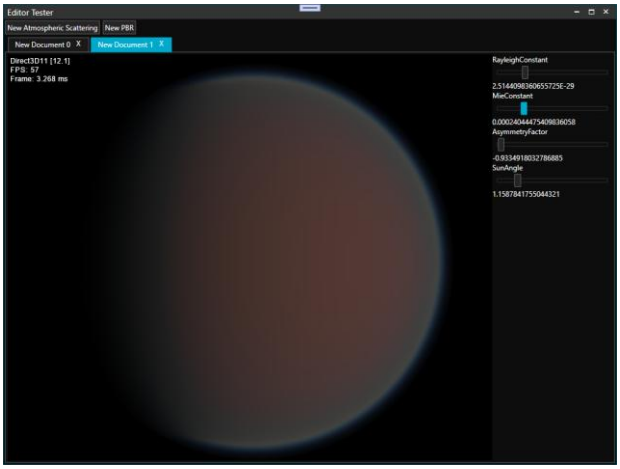


Figure 17 Atmospheric scattering example with medium Rayleigh and Mie constant.

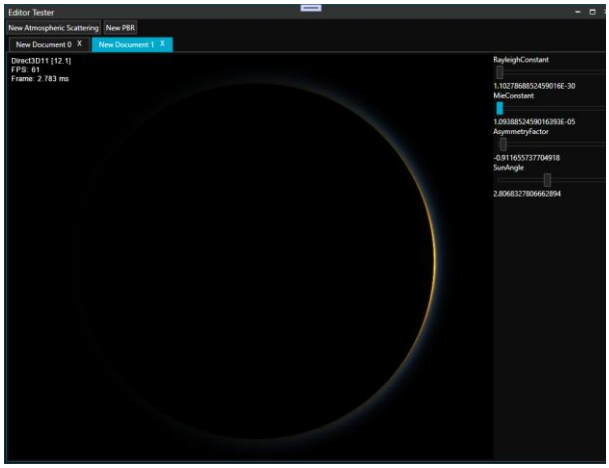


Figure 18 Atmospheric scattering example with low Rayleigh and Mie constant and the sun from behind the planet.

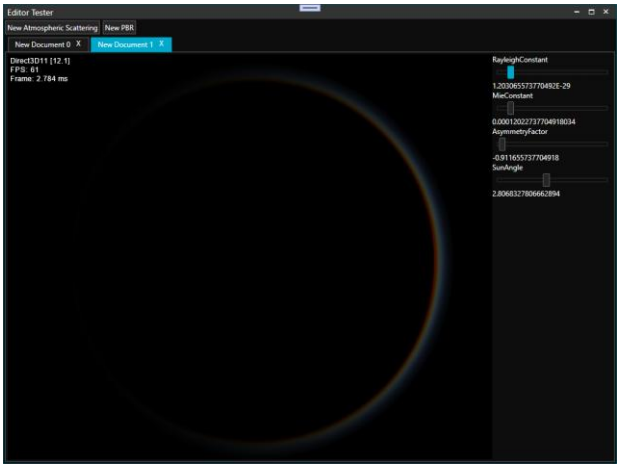


Figure 19 Atmospheric scattering example with medium Rayleigh and Mie constant and the sun from behind the planet.

SHADOW MAPPING

An example of deferred rendering and shadow mapping which uses multiple render targets is illustrated below.

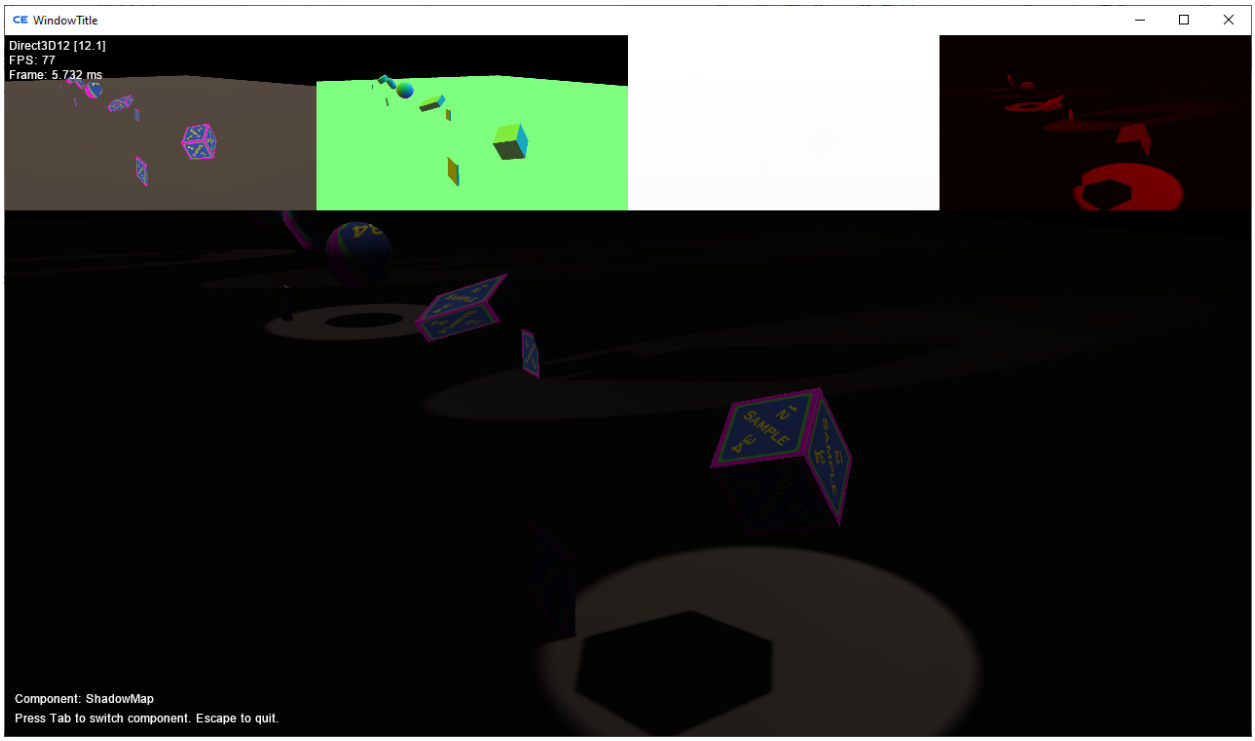


Figure 20 Deferred rendering illustrating a shadow mapping example with the albedo, normal, depth, and light render targets. In the background is the result of the final pass combining all render targets.

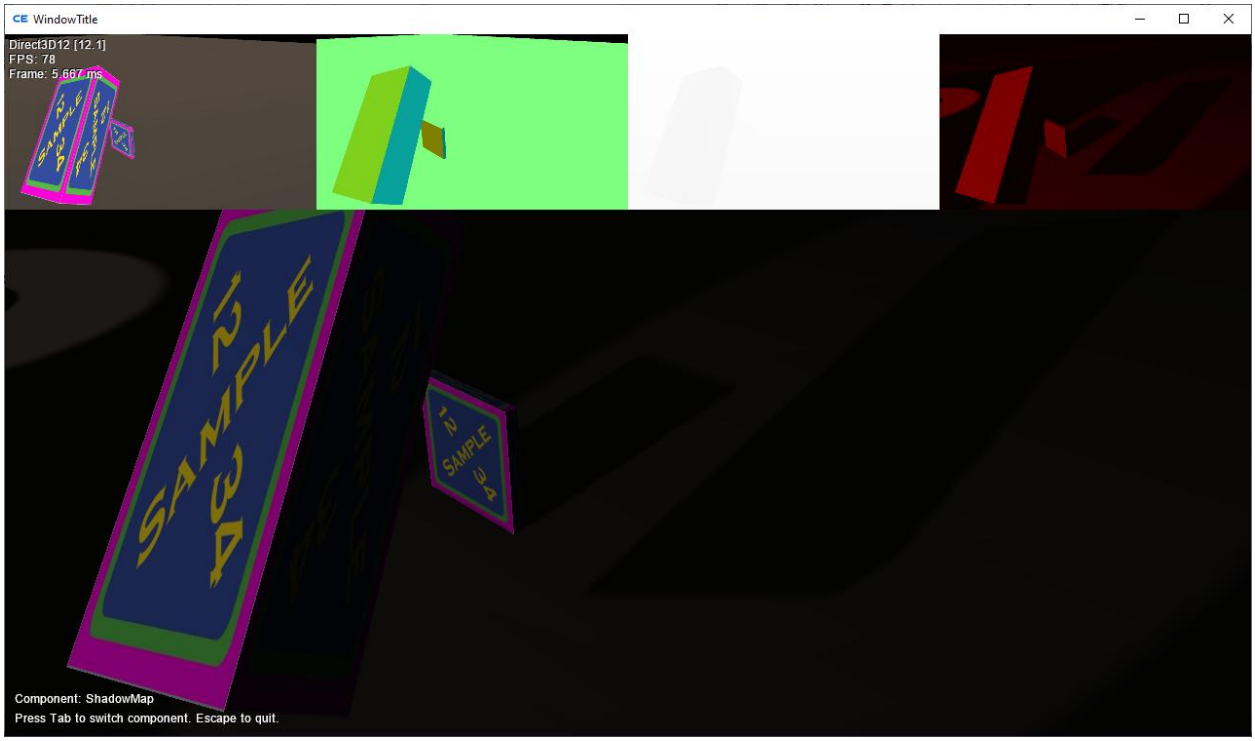


Figure 21 Another image of the shadow mapping example.

SPRITES

CyberEngine uses a sprite batch to render images and text from fonts. The fonts can be created in the content pipeline by supplying the font properties and a texture is automatically generated, or a custom font can be created externally in image editing software.

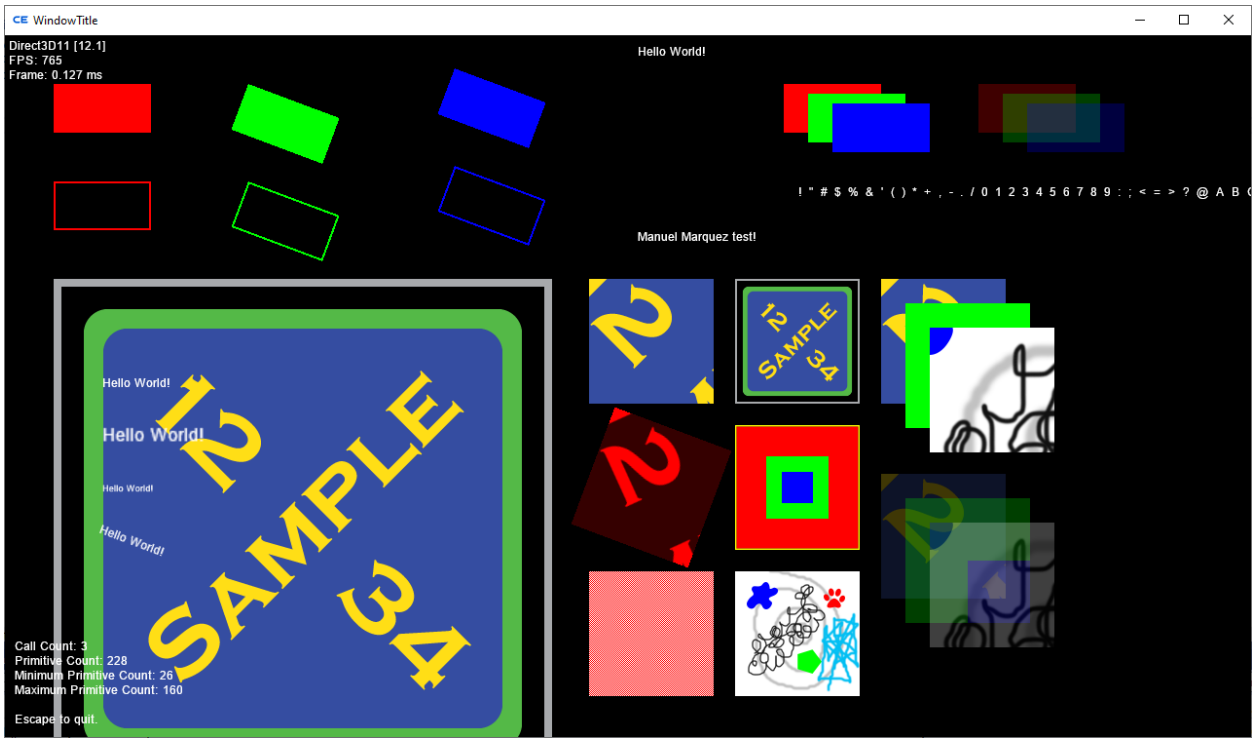


Figure 22 Sprite batch is used to render multiple sprites and text.

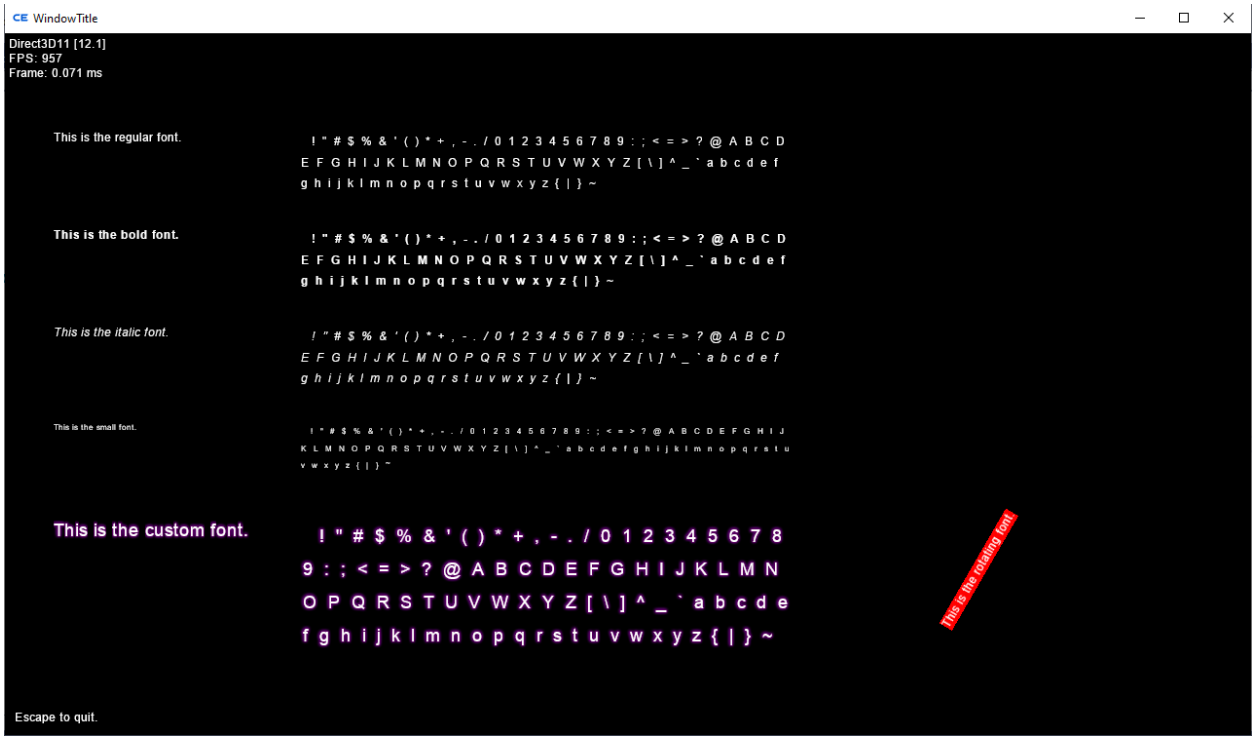


Figure 23 Illustration of rendering text as sprites which are created in the content pipeline or custom designed in image editor software.

Ray Tracing

CyberEngine supports ray tracing with Direct3D 12 and a simple example is illustrated below.

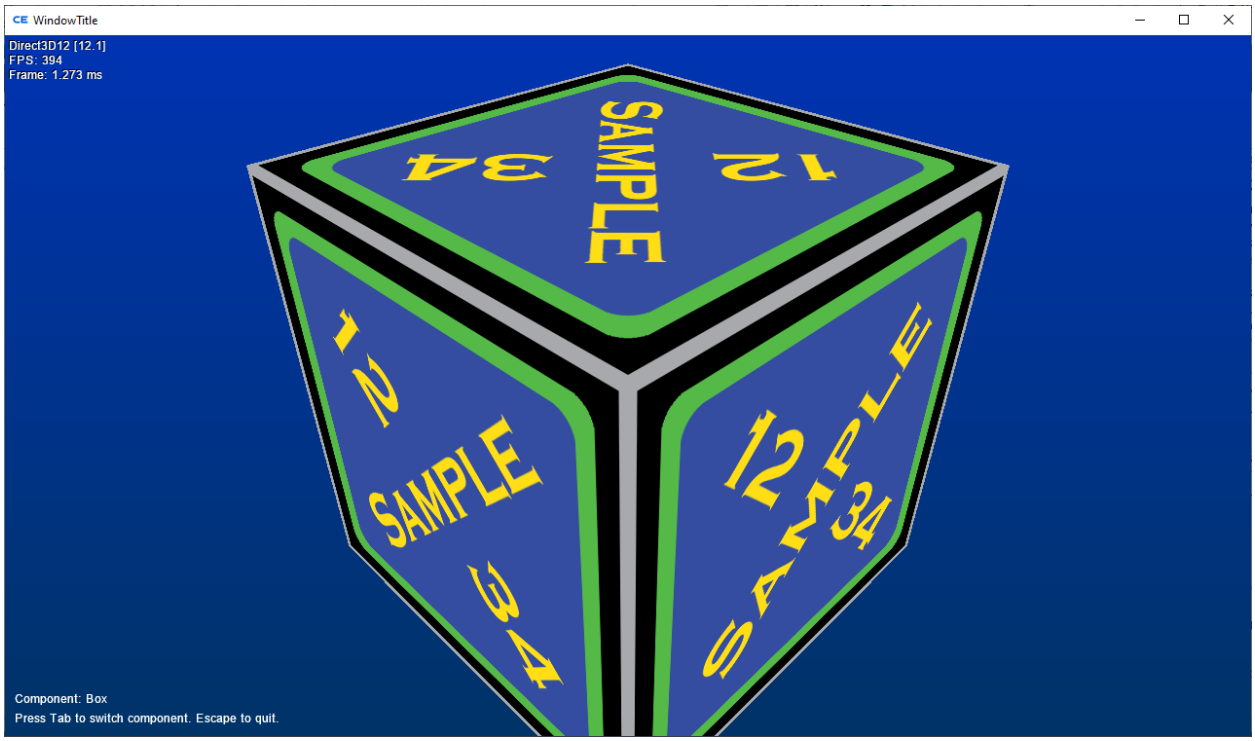


Figure 24 Ray tracing example in Direct3D 12.

CPU Ray Tracing

In order to debug shader concepts, a CPU based ray tracer was architected in C#. A scene can be rendered by supplying geometry and hit and miss functions. A multithreading option can be enabled to speed up the output of an image by dividing rays among threads.

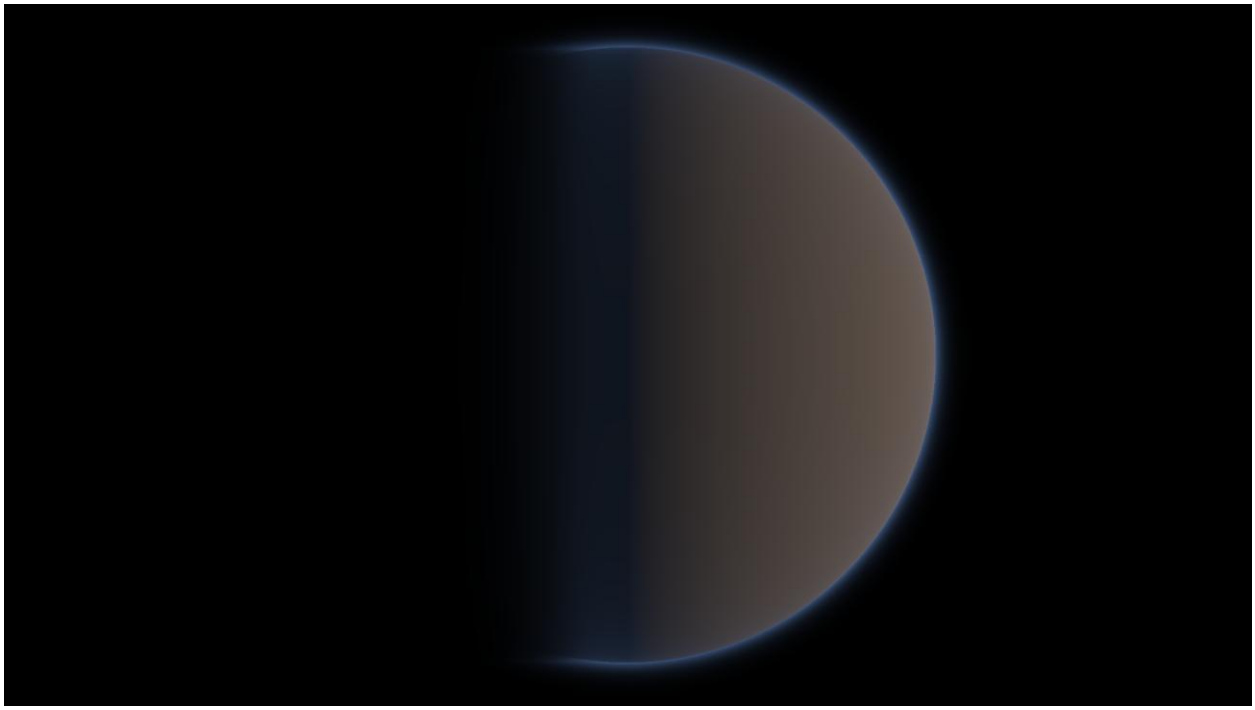


Figure 25 Image of a planet and atmosphere generated from a CPU based ray tracer allowing shader concepts to be written in C#, which has a more robust debugger.

SHADERS

CROSS COMPILER

CyberEngine compiles HLSL shaders for Direct3D and cross compiles into SPIR-V for Vulkan and GLSL for OpenGL. This allows one shader to be written and maintained while supporting all graphics APIs. Shaders use compiler directives to state how buffers should be bound to shader programs. One possible improvement is to automate the assignment of binding indices.

DESCRIPTOR TABLE CODE GENERATOR

CyberEngine auto generates C# code for constant buffers by analyzing the shader code and creating an analogue constant block in C#. The shader is also analyzed to auto generate a descriptor table eliminating the need to manually keep shader code and C# code in sync.

Below is a simple example of a vertex and pixel shader with its auto generated C# code that renders a cube with a texture.

```
/* Includes *****************************************************/
#include "CyberEngine\VertexStructs\PositionTextureVertex.hlsl"
#include "CyberEngine\PixelStructs\PositionTexturePixel.hlsl"


/* Constant buffers *********************************************/
#if OPENGL
[[vk::binding(0)]]
#elif VULKAN
[[vk::binding(0, 0)]]
#endif
cbuffer WorldViewProjectionConstantBlock : register(b0)
{
    // World, view and projection matrix
    row_major float4x4 WorldViewProjection;
};


/* Vertex shader ************************************************/
PositionTexturePixel VS(PositionTextureVertex vertex)
{
    PositionTexturePixel pixel = (PositionTexturePixel) 0;

    // Transform the position to screen space
    pixel.Position = mul(vertex.Position, WorldViewProjection);

    // Set the properties
    pixel.TextureCoordinate = vertex.TextureCoordinate;

    return pixel;
}
```

Figure 26 Vertex shader used to render a cube.

```
// Auto Generated File. Do not modify.

using System;
using CyberneticGames.CyberEngine.Data;
using CyberneticGames.CyberEngine.Graphics;
using CyberneticGames.CyberEngine.Mathematics;

namespace CoreTester.Graphics
{
    public partial class MeshGraphicsProgram
    {

        public static readonly DescriptorTable VertexDescriptorTable = new DescriptorTable(ShaderStage.Vertex,
            new[]
            {
                new DescriptorElement(DescriptorType.ConstantBuffer),
            });

        public static void SetWorldViewProjectionConstantBlock(IDescriptorBinding descriptorBinding,
                                                                int tableIndex, IConstantBuffer buffer)
                                => descriptorBinding.SetConstantBuffer(tableIndex, 0, buffer);


        public class WorldViewProjectionConstantBlock : ConstantBlock
        {
            private static readonly IConstantValueLayout[] ValueLayouts = new IConstantValueLayout[]
            {
                new
ConstantValueLayout<CyberneticGames.CyberEngine.Mathematics.MatrixF>(nameof(WorldViewProjection)),
            };

            public WorldViewProjectionConstantBlock(GraphicsEngine graphicsEngine)
                : base(ValueLayouts, graphicsEngine)
            {
            }

            public CyberneticGames.CyberEngine.Mathematics.MatrixF WorldViewProjection { set
                        => Data.SetValue(nameof(WorldViewProjection), value); }
        }

    }
}
```

Figure 27 Auto generated descriptor table and constant block for the previous vertex shader. Any changes made to the shader will automatically regenerate the C# code.

```
/* Includes ************************************************/
#include "CyberEngine\PixelStructs\PositionTexturePixel.hlsl"


/* Constants ***********************************************/

// Alpha channel clip threshold
static const float ClipThreshold = 0.001;


/* Textures and samplers ***********************************************/
#if OPENGL
[[vk::binding(0)]]
#elif VULKAN
[[vk::binding(0, 1)]]
#endif
Texture2D<float4> Texture : register(t0);

#if VULKAN
[[vk::binding(1, 1)]]
#endif
SamplerState TextureSampler : register(s0);


/* Pixel shader ********************************************/
float4 PS(PositionTexturePixel pixel) : SV_Target0
{
    // Get the colour
    float4 colour = Texture.Sample(TextureSampler, pixel.TextureCoordinate);

    // Clip pixels that are transparent
    clip(colour.a - ClipThreshold);

    return float4(colour.rgb, 1);
}
```

Figure 28 Pixel shader used to render a cube.

```
// Auto Generated File. Do not modify.

using System;
using CyberneticGames.CyberEngine.Data;
using CyberneticGames.CyberEngine.Graphics;
using CyberneticGames.CyberEngine.Mathematics;

namespace CoreTester.Graphics
{
    public partial class MeshGraphicsProgram
    {

        public static readonly DescriptorTable PixelDescriptorTable = new DescriptorTable(ShaderStage.Pixel,
            new[]
            {
                new DescriptorElement(DescriptorType.SampledTexture),
            });

        public static void SetTexture(IDescriptorBinding descriptorBinding, int tableIndex, ITextureView texture,
                                      ITextureSampler textureSampler)
                                => descriptorBinding.SetSampledTexture(tableIndex, 0, texture, textureSampler);


    }
}
```

Figure 29 Auto generated descriptor table and method to the set the texture and texture sampler. Any changes made to the shader will automatically regenerate the C# code.

GEOMETRY SHADER

CyberEngine supports vertex, geometry, pixel, and compute shaders.

Figure 30 Example of a geometry shader rendering the normals of the points and face of triangles.

COMPUTE SHADER

An example of a compute shader that generates the visible geometry from a 3D voxel texture.

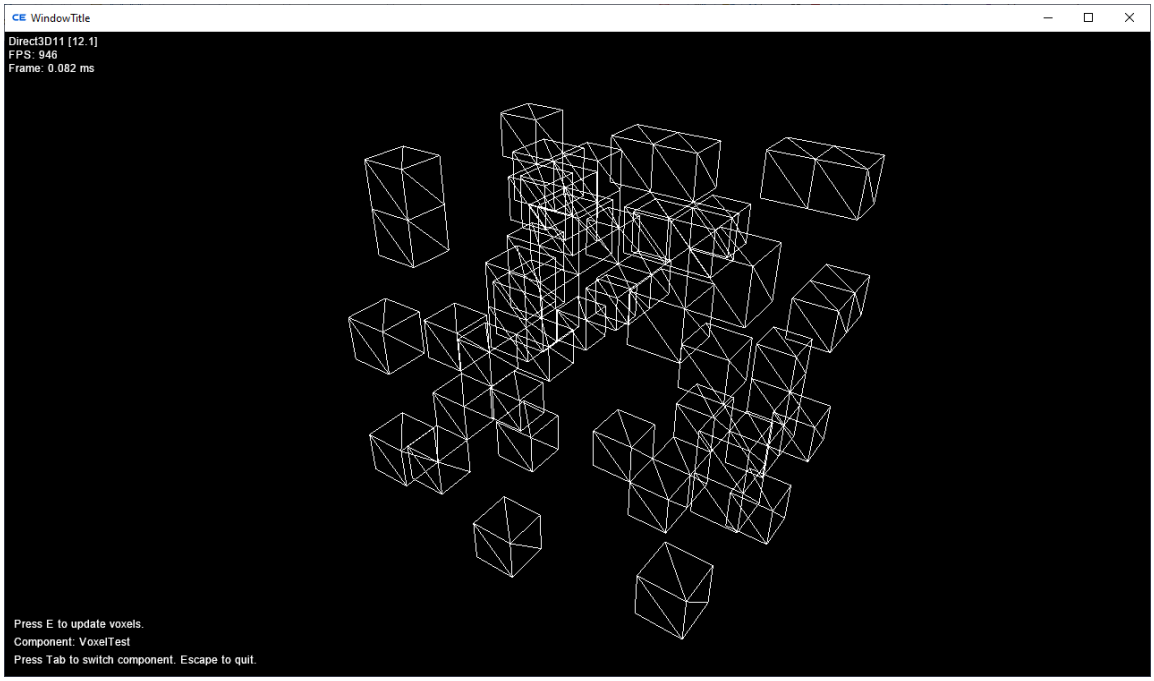


Figure 31 Compute shader generating visible geometry from a 3D voxel texture.

An example of a compute shader executed in a console application to create a noise texture and save to an image.

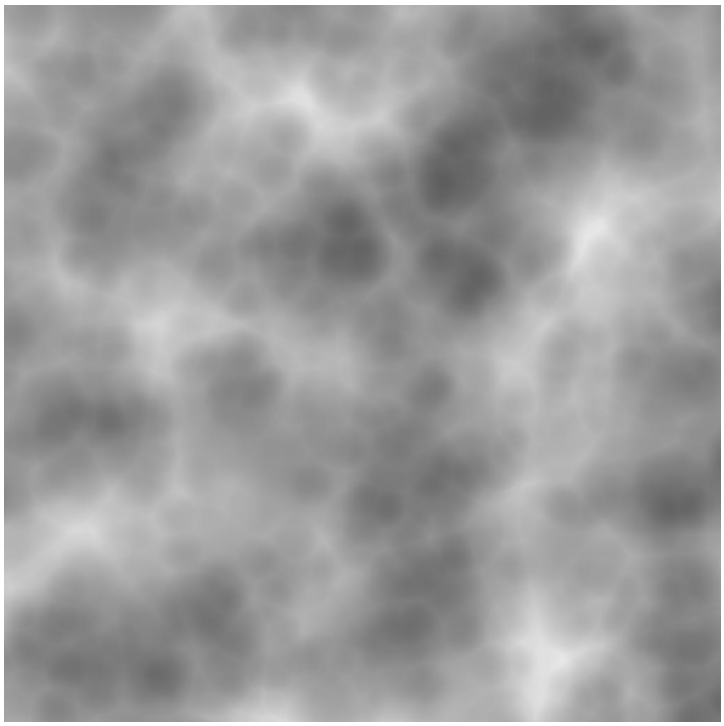


Figure 32 Voronoi noise texture generated on the GPU.

User Interface

A user interface framework inspired by WPF created from scratch, supports controls, control styles, and data binding. The user interface is rendered through a drawing interface allowing it to be decoupled from the graphics APIs including Direct3D 11, Direct3D 12, OpenGL, and Vulkan. This also enabled the ability to render to an image, allowing automated tests to be debugged by viewing the results of the UI.

Styling

An example of a Button style is shown below. It illustrates setting default properties and changing properties on triggers.

```
<c:Style TargetType="{c:Type Button}">
  <c:Setter Property="Button.Margin" Value="3" />
  <c:Setter Property="Button.Padding" Value="2" />

  <c:Setter Property="TextBlock.Foreground" Value="#ffffff" />
  <c:Setter Property="Button.Background" Value="#262626" />
  <c:Setter Property="Button.BorderBrush" Value="#666666" />
  <c:Setter Property="Button.BorderThickness" Value="1" />

  <c:Setter Property="Button.Template">
    <c:Setter.Value>
      <c:Template>
        <Border Background="{c:TemplateBinding Background}" BorderBrush="{c:TemplateBinding BorderBrush}"
                BorderThickness="{c:TemplateBinding BorderThickness}">
          <ContentPresenter Margin="{c:TemplateBinding Padding}" />
        </Border>
      </c:Template>
    </c:Setter.Value>
  </c:Setter>

  <c:Style.Triggers>
    <c:PropertyTrigger Property="Button.IsMouseWithin" Value="True">
      <c:Setter Property="TextBlock.Foreground" Value="#ffffff" />
      <c:Setter Property="Button.Background" Value="#1eb300" />
      <c:Setter Property="Button.BorderBrush" Value="#158000" />
    </c:PropertyTrigger>

    <c:PropertyTrigger Property="Button.IsPressed" Value="True">
      <c:Setter Property="TextBlock.Foreground" Value="#ffffff" />
      <c:Setter Property="Button.Background" Value="#158000" />
      <c:Setter Property="Button.BorderBrush" Value="#116600" />
    </c:PropertyTrigger>

    <c:PropertyTrigger Property="Button.IsEnabled" Value="False">
      <c:Setter Property="TextBlock.Foreground" Value="#999999" />
      <c:Setter Property="Button.Background" Value="#1a1a1a" />
      <c:Setter Property="Button.BorderBrush" Value="#333333" />
    </c:PropertyTrigger>
  </c:Style.Triggers>
</c:Style>
```

Figure 33 Button style.

Data Binding

An example of how data binding is configured is shown below. Data binding relies on reflection to resolve properties. Events can also be hooked up to the code file by name, or can be bound to a view model command adhering to the MVVM design pattern.

```
<TabItem Header="Button">
  <StackPanel>
    <TextBlock Name="ButtonTextBlock" />

    <Button Click="Button_Click">
      <TextBlock Text="Button" />
    </Button>

    <RepeatButton Click="RepeatButton_Click">
      <TextBlock Text="RepeatButton" />
    </RepeatButton>

    <ToggleButton Name="ToggleButton">
      <TextBlock Text="ToggleButton" />
    </ToggleButton>

    <TextBlock Text="{c:Binding Source={c:NamedElementBindingSource ToggleButton}, IsChecked}" />

    <TextBlock Text="Text to show and hide with toggle button"
               Visibility="{c:Binding Source={c:NamedElementBindingSource ToggleButton}, IsChecked,
                           Converter={c:Resource BooleanToVisibilityConverter}}" />
```

Figure 34 Portion of UI for the buttons illustrating the link to actionable events and data binding.

EXAMPLES

Below are a few screens from a test application showing some of the controls.

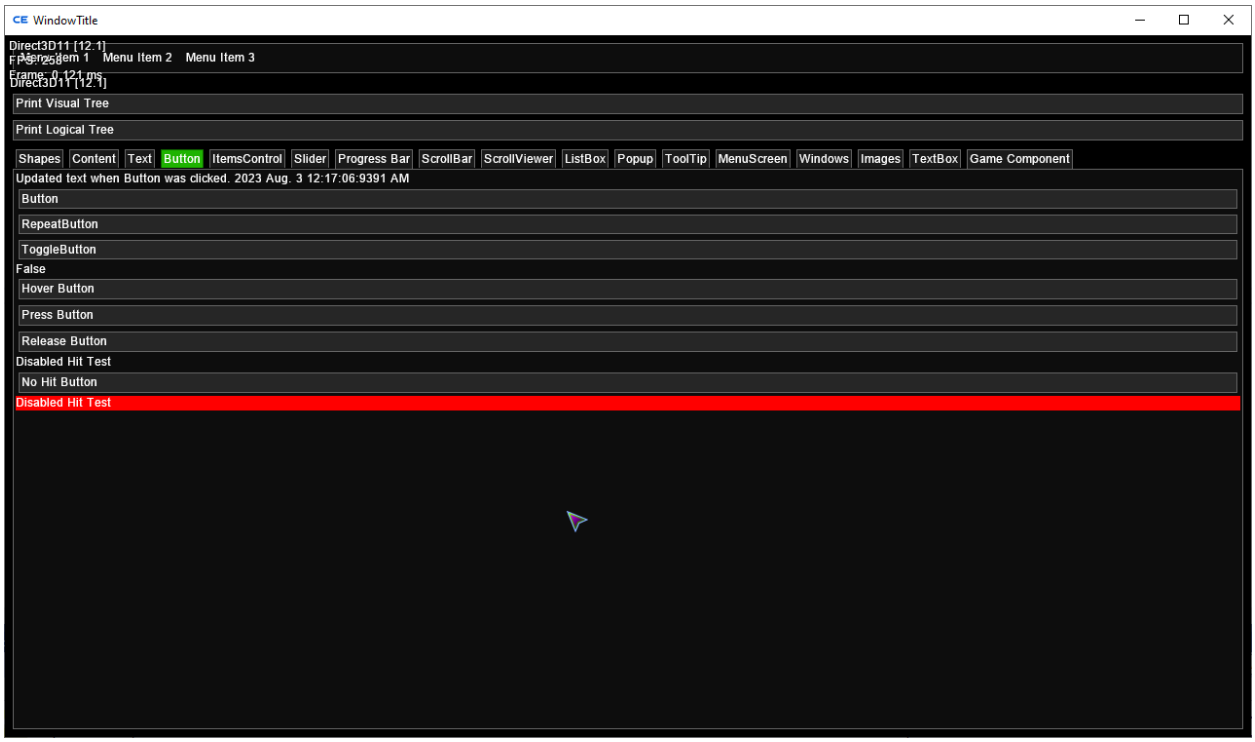


Figure 35 Part of a test user interface showing a tab control with the buttons currently selected.

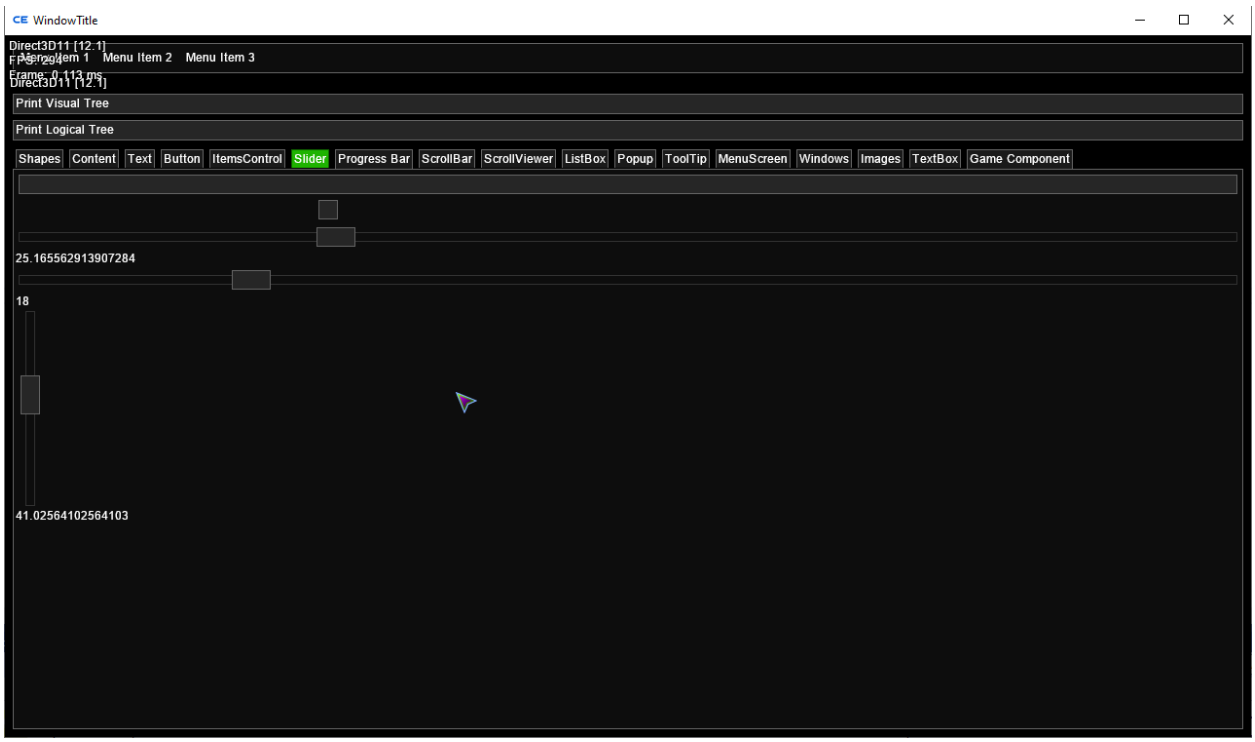


Figure 36 Illustration of the sliders tab with a binding for the slider value to the text block.

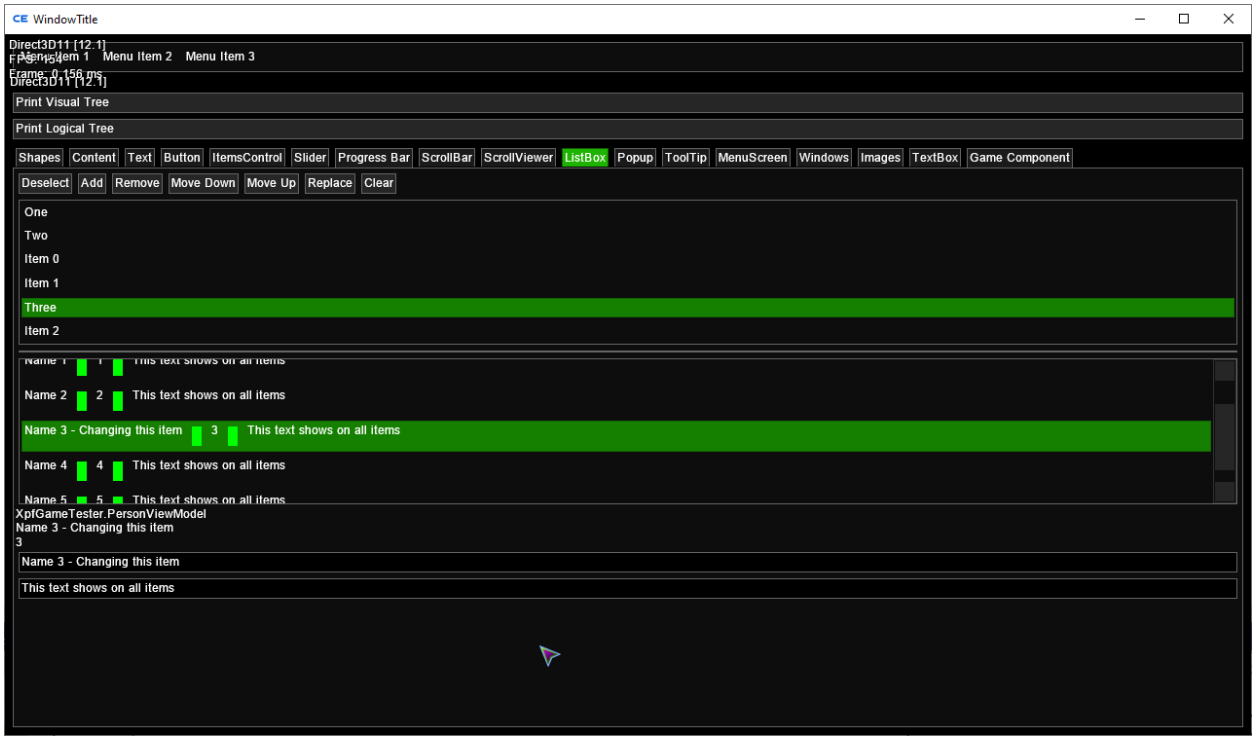


Figure 37 List control allowing the selection of items and command buttons that perform actions on the list items. The lower portion is also bound to the selected item and can be edited with a text box.

Figure 38 3D viewport within the user interface.

3D UI

The user interface can be drawn to a render target and later rendered on to a 3D object. User input is transformed into the 3D scene so it can be interacted with.

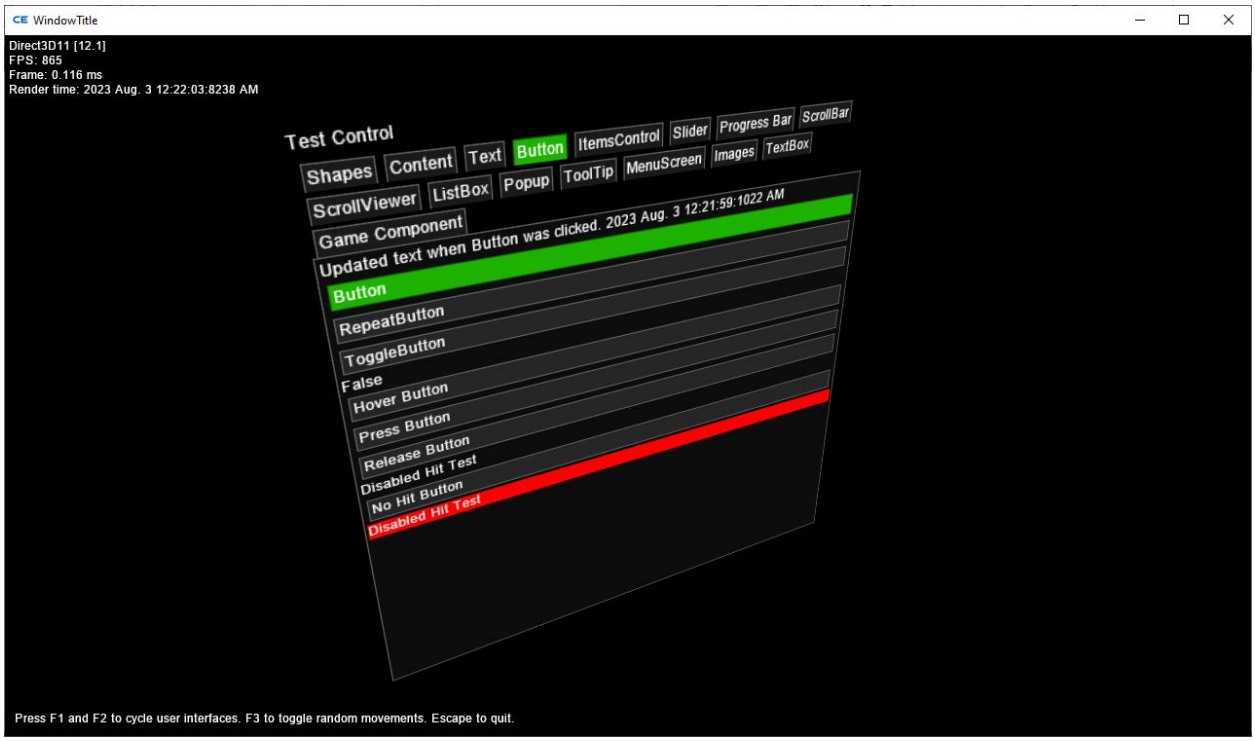


Figure 39 User interface that is rendered onto a 3D quad and also is intractable.

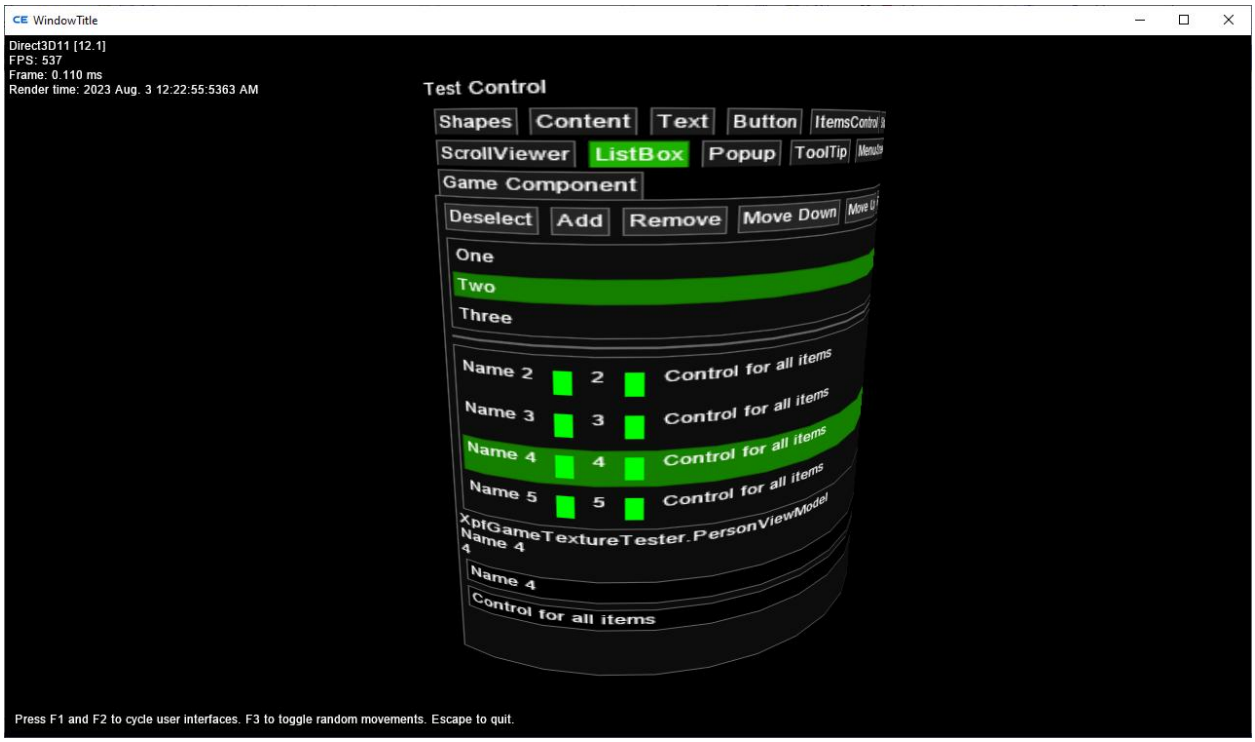


Figure 40 User interface rendered onto a half cylinder.

WPF

A game control for WPF allows scenes to be hosted in WPF applications. Support exists for all graphics APIs. Since WPF runs on Direct3D 9, render targets must be copied to a Direct3D 9 texture for display in WPF applications.

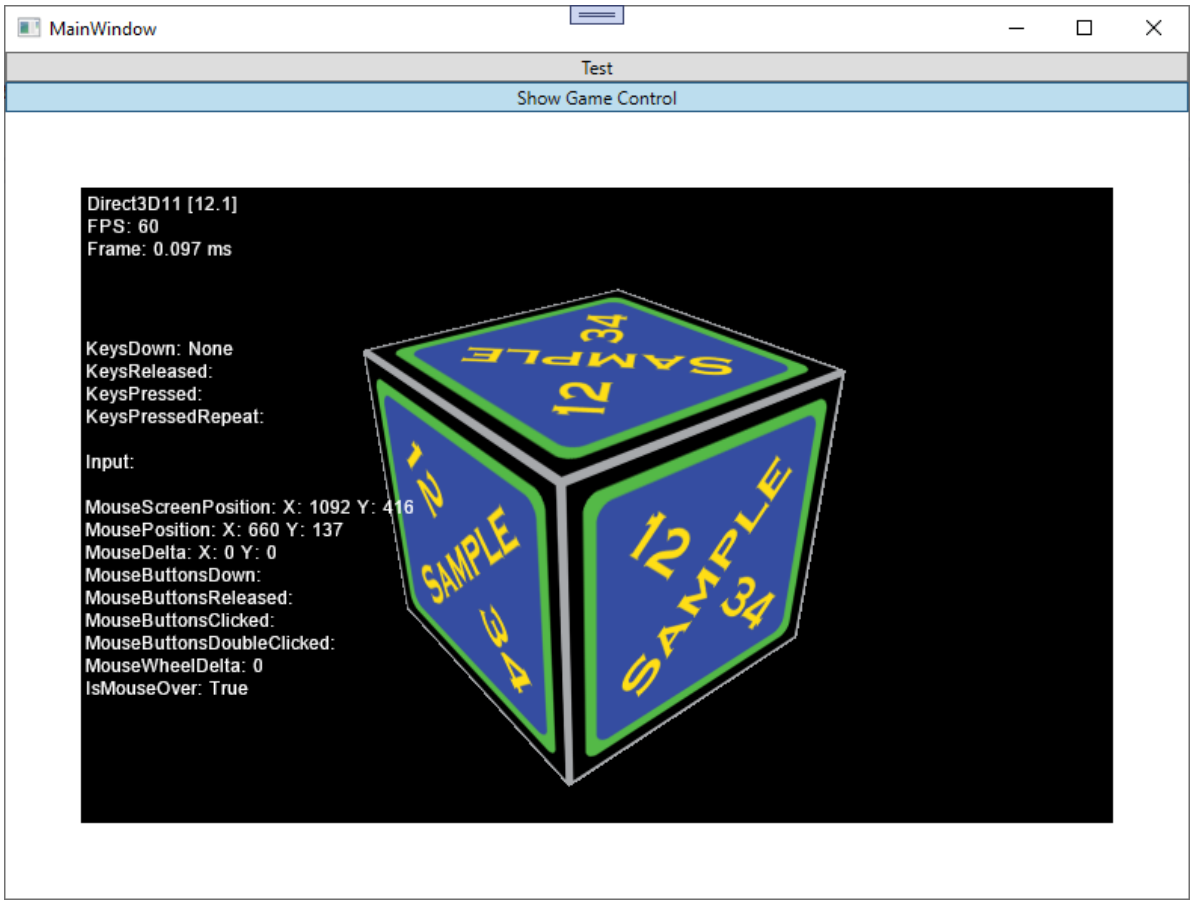


Figure 41 WPF component to render a scene within a WPF application. Direct3D 11 illustrated in this example.

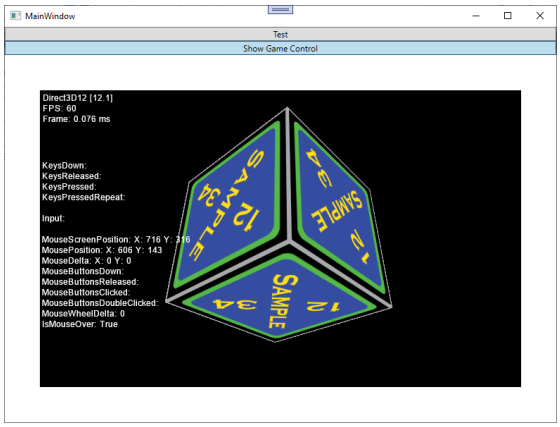


Figure 42 Same WPF example rendered with Direct3D 12.

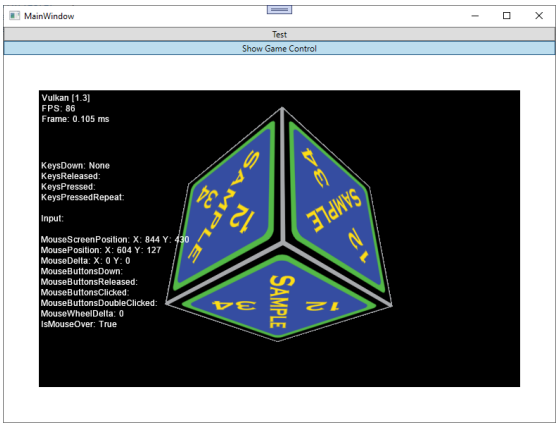


Figure 43 Same example rendered in Vulkan.

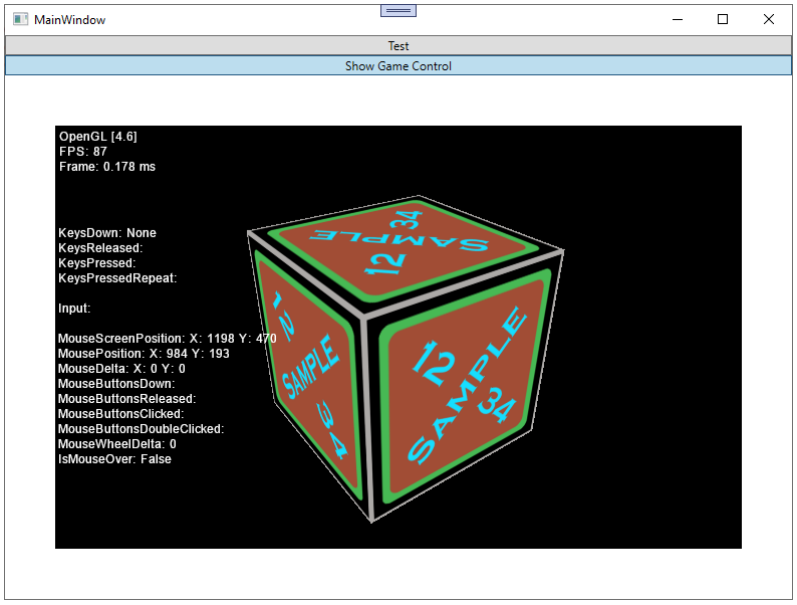


Figure 44 Same WPF example rendered with OpenGL. This example has incorrect colours due to differing render target formats with OpenGL and WPF. RGBA and BGRA are swapping the red and blue channels. This can be seen as the grey and green appear correct, but the blue and yellow are not.

PHYSICS

CyberEngine contains a constraint based physics engine. Narrow phase collision detection uses optimized algorithms between common shapes and GJK, which can be extended to support custom convex hulls. Collision response uses an iterative solver that supports contact manifolds and static and dynamic friction.

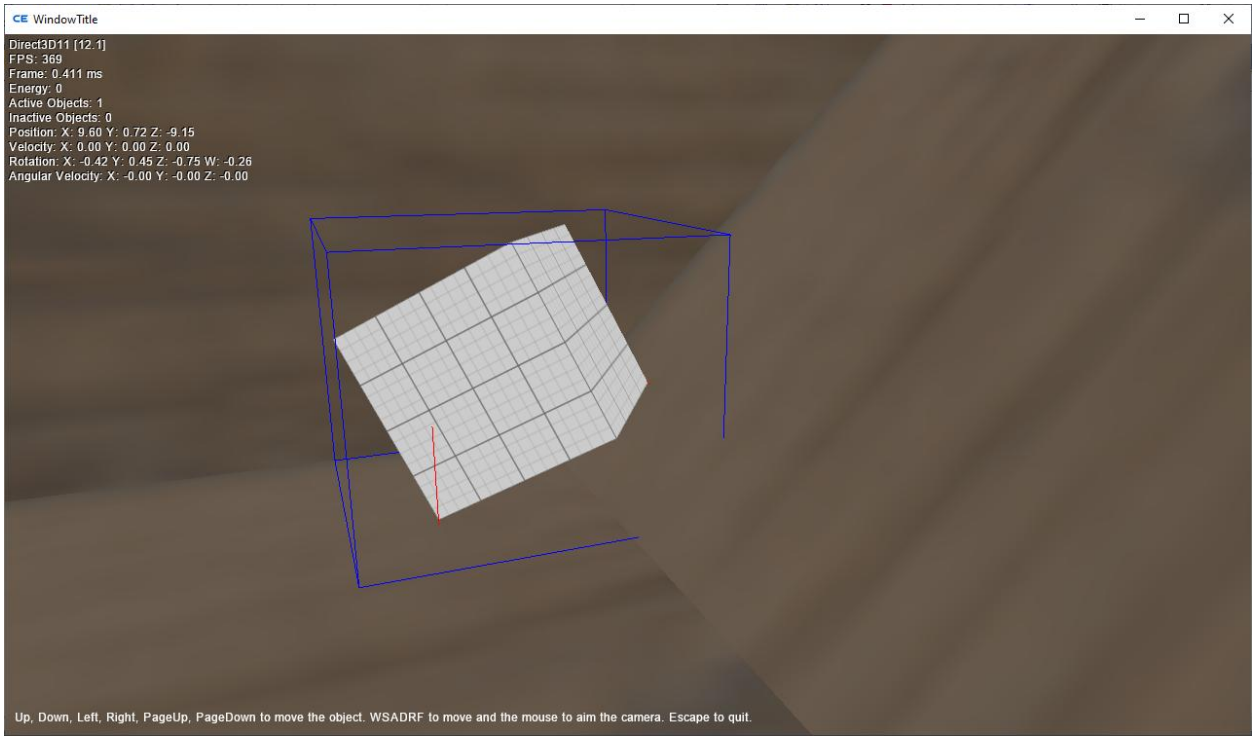


Figure 45 Cube within a physics engine debugger illustrating the AABB and contact normals.

Models

CyberEngine supports 3D models and animation. The content pipeline supports Collada and GLTF files. A test application to view models and animations is illustrated below.

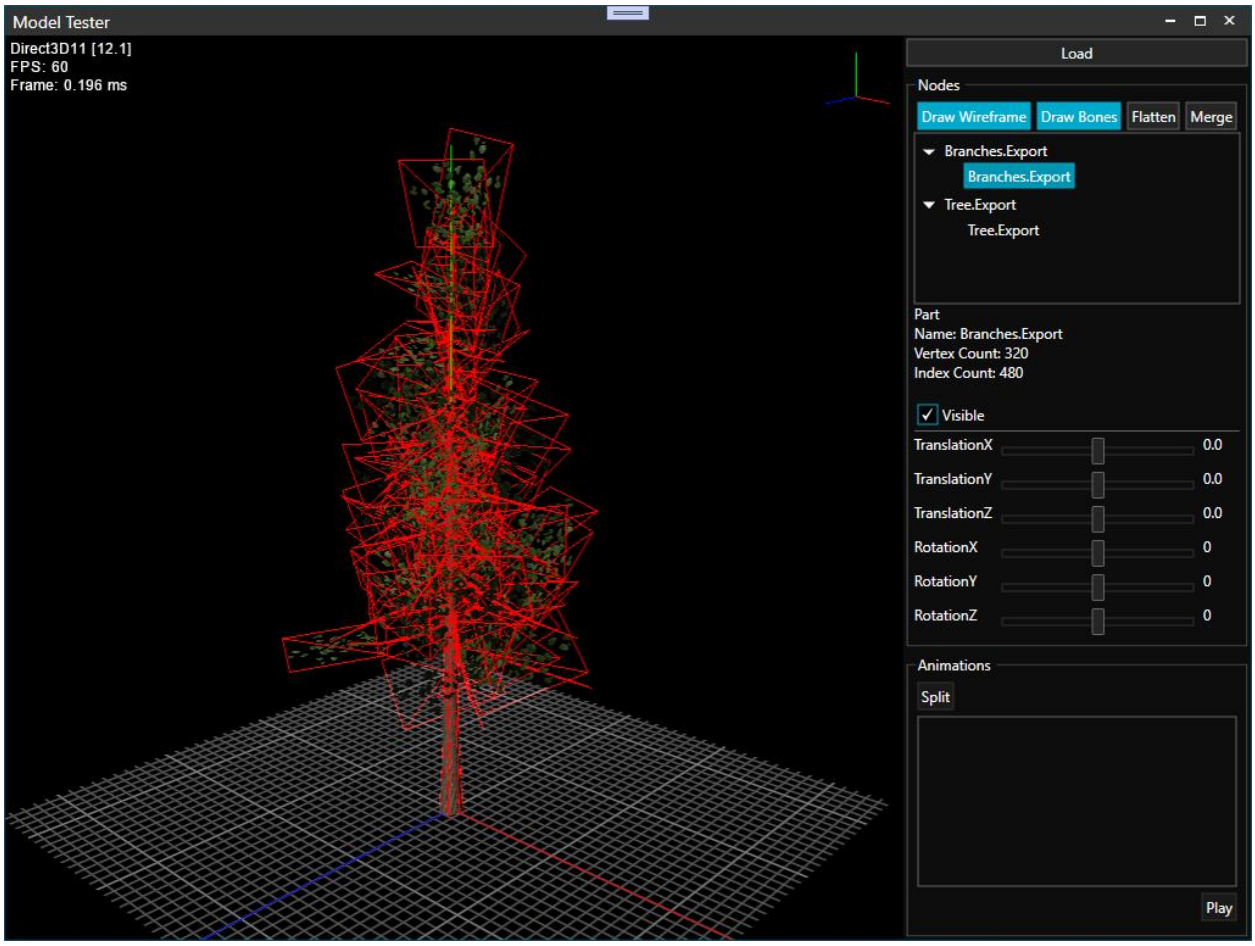


Figure 46 3D model renderer hosted in a WPF application.

Instanced Animation

Multiple skinned model instances with bone vertex weights can be rendered in a single draw call using instanced geometry and an array of bone transforms for each instance.

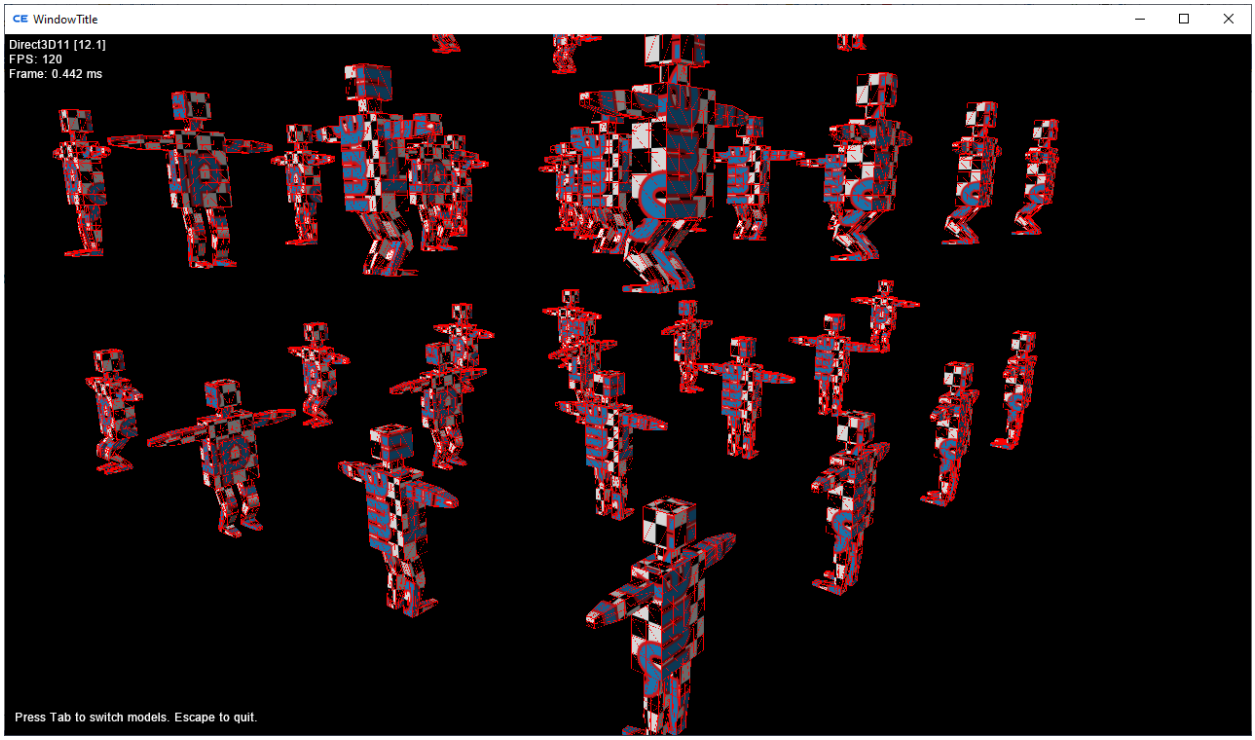


Figure 47 Instanced skeleton model rendering where each model has its own skeletal animation.

LINUX

CyberEngine supports Linux using OpenGL and Vulkan.

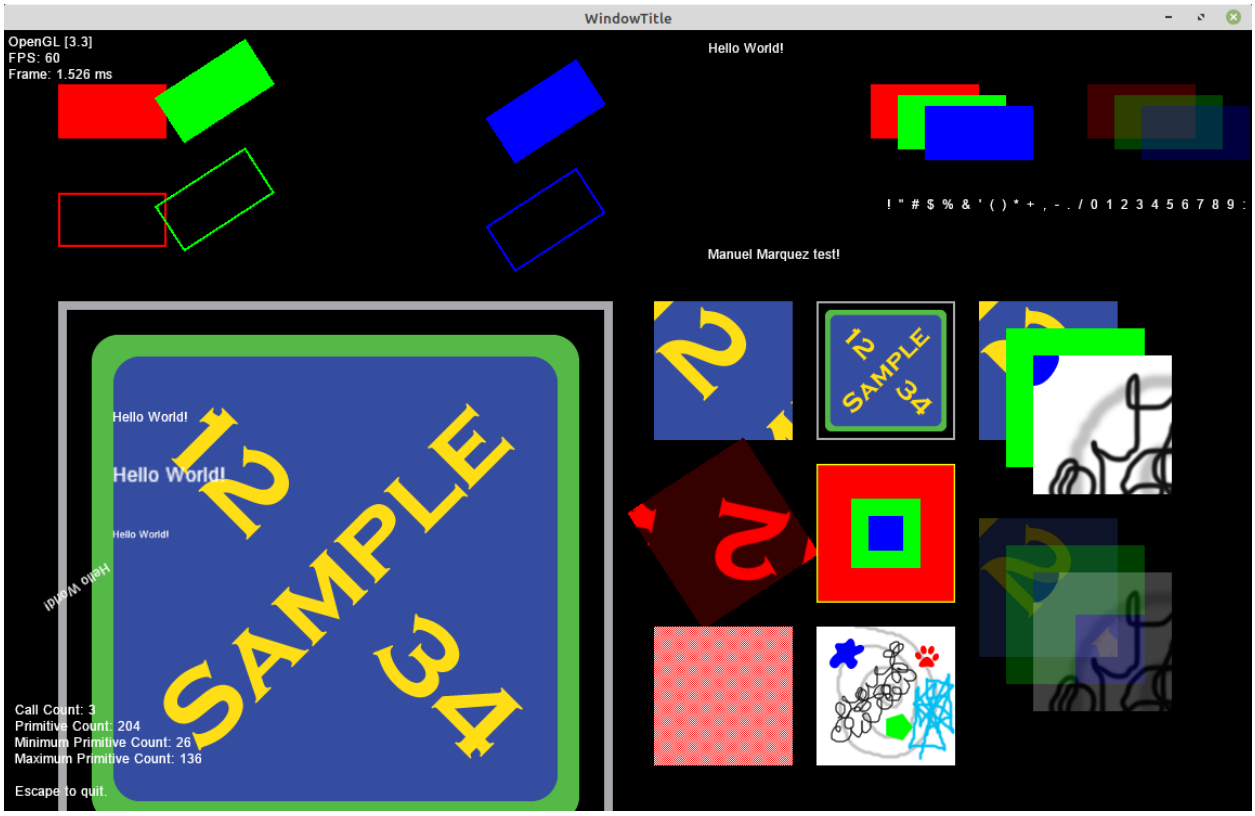


Figure 48 Sprite example captured running on Linux Mint within a virtual machine using OpenGL.

Visual Studio

Project System

CyberEngine uses a Visual Studio extension to provide a custom project type for game content. The project system is built on MSBuild, so it can be supported in Visual Studio and integrate with C# projects that also use MSBuild. It allows content files to be added to a project and configure importers to handle processing and writing the compiled content file.

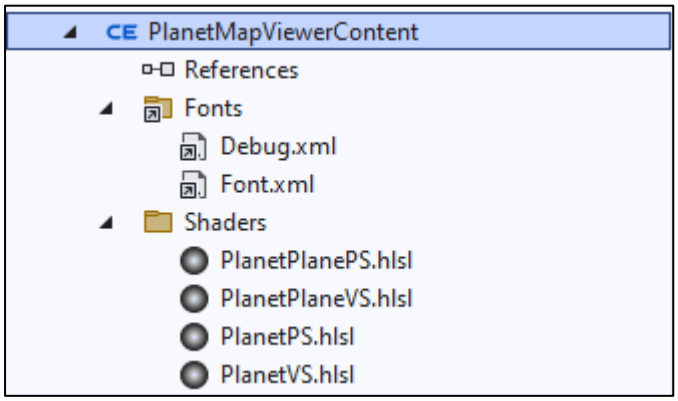


Figure 49 Custom game content project type within the Visual Studio Solution Explorer.

Project File

During game execution, the file is read which contains the reader type name to properly parse and load the content. The importers are decoupled from the readers so the writers do not need to be shipped with the game.

```
<ItemGroup>
  <PackageReference Include="CyberneticGames.CyberEngine.Content.Pipeline" Version="1.*" />
  <PackageReference Include="CyberneticGames.CyberEngine.Content.Pipeline.Direct3D11" Version="1.*" />
  <PackageReference Include="CyberneticGames.CyberEngine.Graphics" Version="1.*" />
</ItemGroup>
<ItemGroup>
  <Content Include="Shaders\MeshVS.hlsl">
    <Importer_Type>CyberneticGames.CyberEngine.Content.Pipeline.CrossShader.VertexShaderImporter</Importer_Type>
    <Importer_EntryPoint>VS</Importer_EntryPoint>
    <Importer_Profile>vs_4_0</Importer_Profile>
    <Importer_EnableDebugging Condition="'$(Configuration)' == 'Debug'">True</Importer_EnableDebugging>
    <Importer_Direct3D11Version>4.0</Importer_Direct3D11Version>
    <Importer_Direct3D12Version>4.0</Importer_Direct3D12Version>
    <Importer_OpenGLVersion>330</Importer_OpenGLVersion>
    <Importer_VulkanVersion>450</Importer_VulkanVersion>
  </Content>
  <Content Include="Shaders\MeshPS.hlsl">
    <Importer_Type>CyberneticGames.CyberEngine.Content.Pipeline.CrossShader.PixelShaderImporter</Importer_Type>
    <Importer_EntryPoint>PS</Importer_EntryPoint>
    <Importer_Profile>ps_4_0</Importer_Profile>
    <Importer_EnableDebugging Condition="'$(Configuration)' == 'Debug'">True</Importer_EnableDebugging>
    <Importer_Direct3D11Version>4.0</Importer_Direct3D11Version>
    <Importer_Direct3D12Version>4.0</Importer_Direct3D12Version>
    <Importer_OpenGLVersion>330</Importer_OpenGLVersion>
    <Importer_VulkanVersion>450</Importer_VulkanVersion>
  </Content>
</ItemGroup>
```

Figure 50 Excerpt from a custom game content project illustrating how assets are configured. NuGet packages are added to the project that contain the content pipeline assemblies.

NuGet

The importers are contained in assemblies and packaged into NuGet packages which can be added to any game content project. The project type in Visual Studio supports using the NuGet package manager to manage NuGet packages for the project.

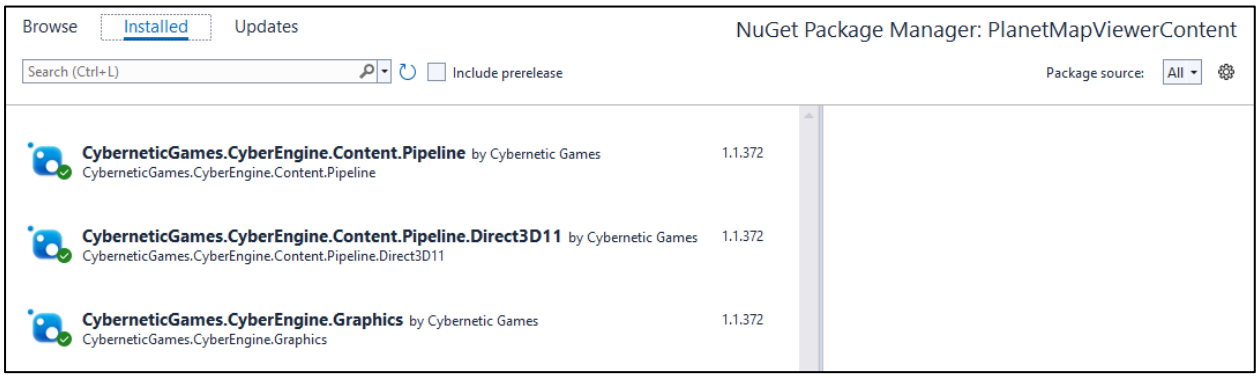


Figure 51 Captured within Visual Studio illustrating full NuGet support for the custom game content project type within the IDE.

Vegabot

Vegabot is a game built on CyberEngine that includes an orbital mechanics simulator. A few images captured from tools and test applications are shown below.

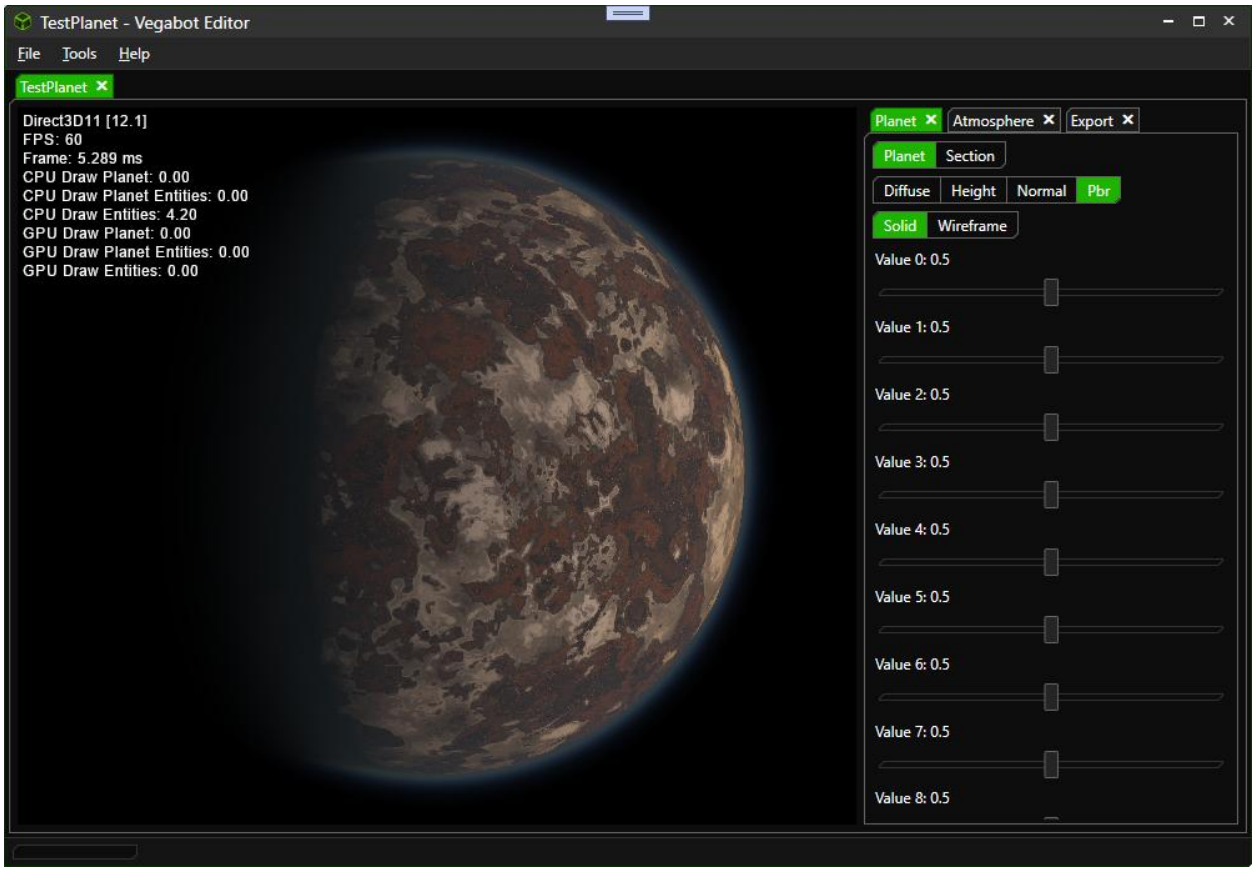


Figure 52 Planet editor built within WPF.

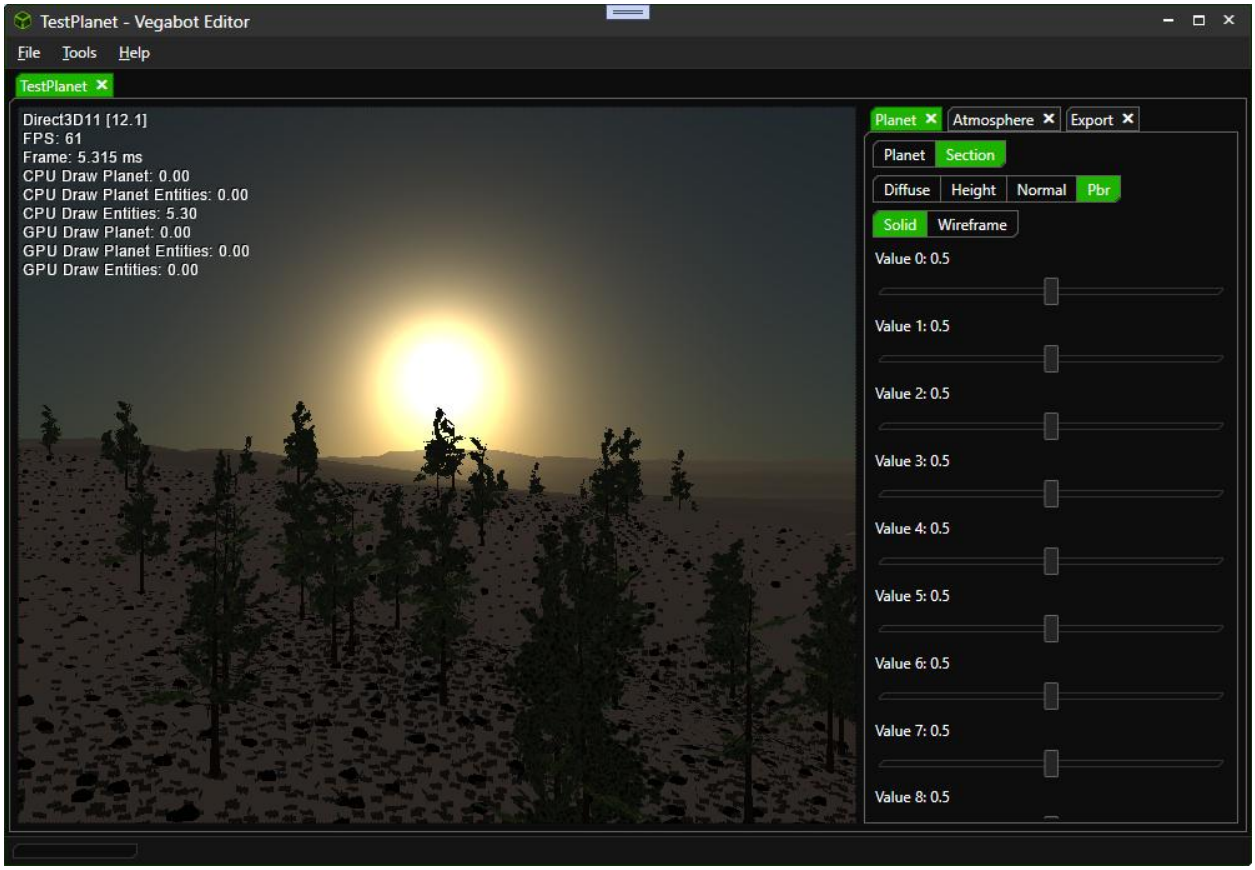


Figure 53 View closer to a planet surface within the editor.

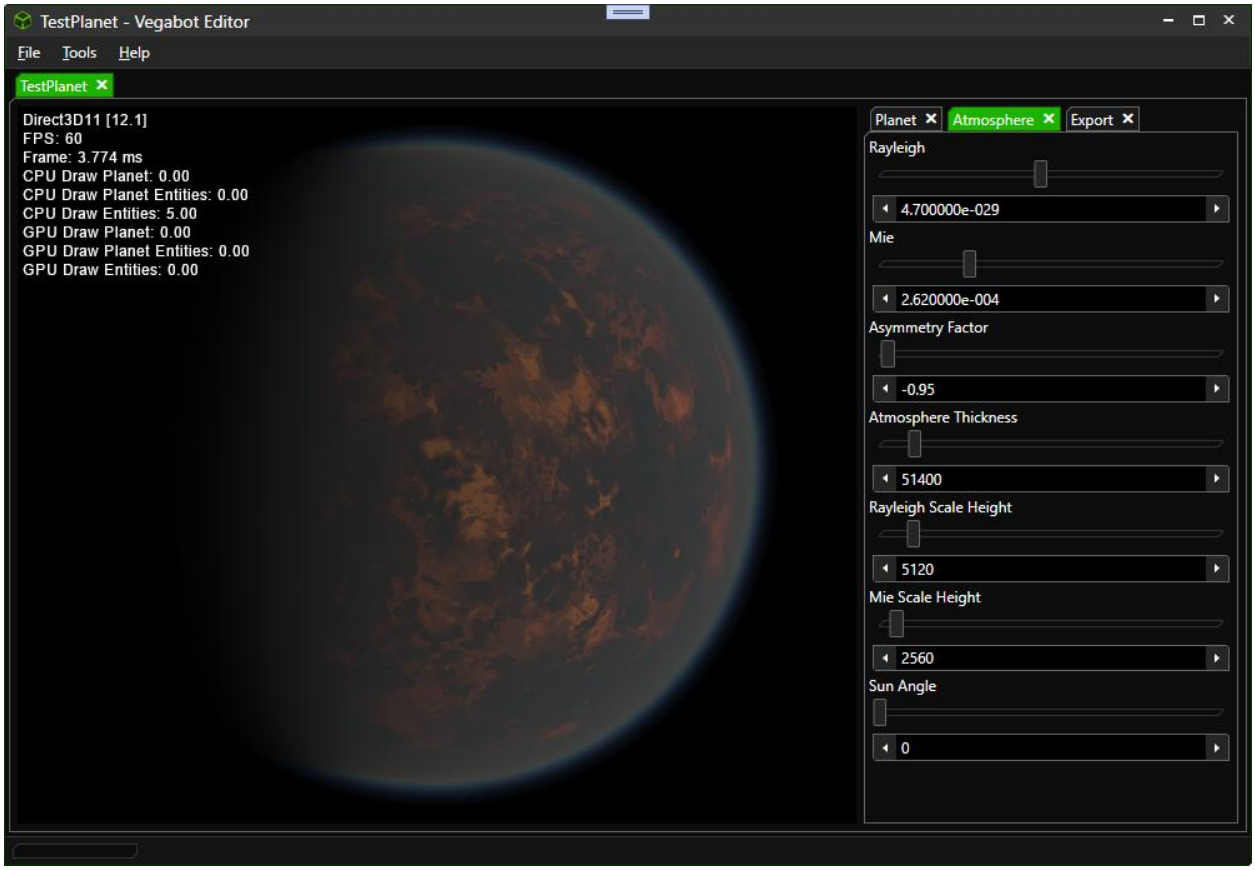


Figure 54 Planet with increased Rayleigh and Mie scattering coefficients.

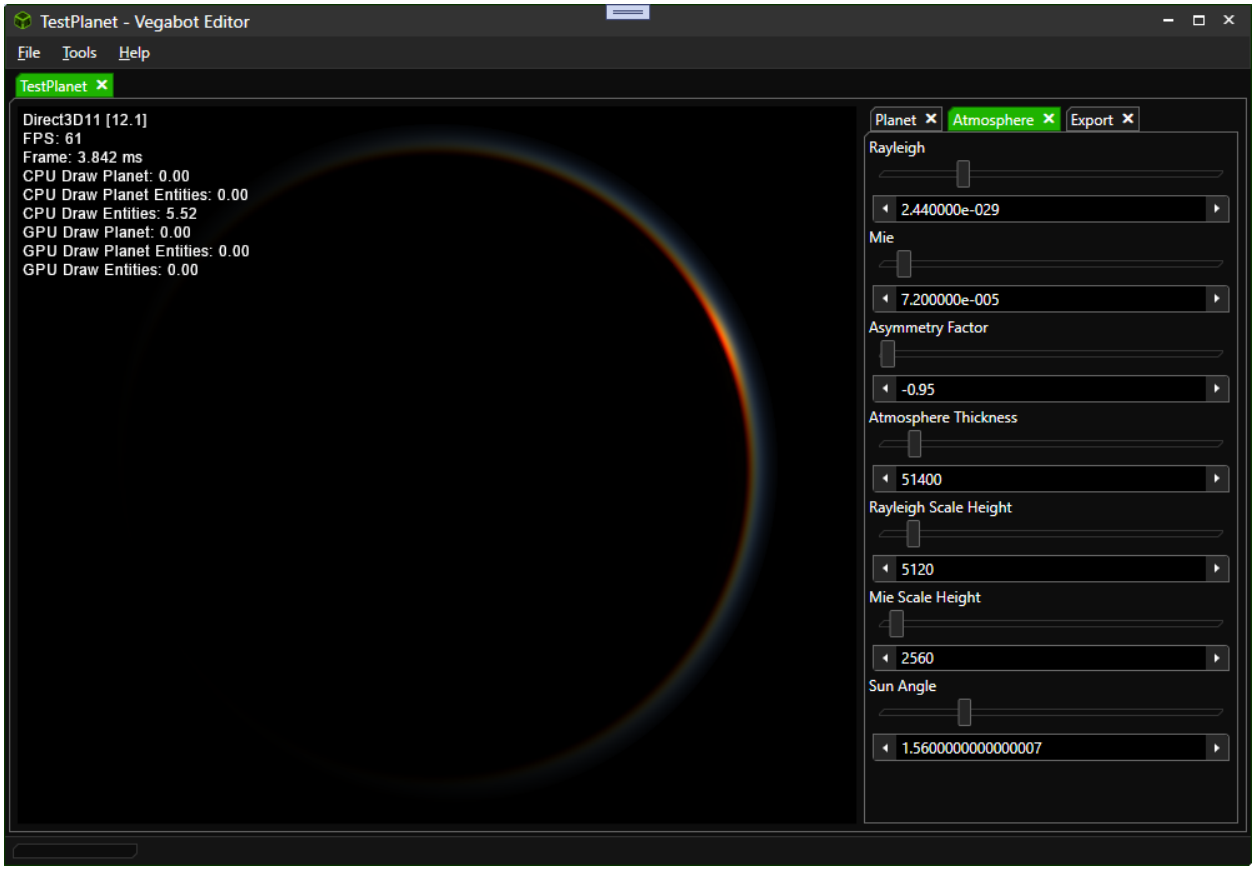


Figure 55 View of planet atmospheric scattering with the sun behind a planet.

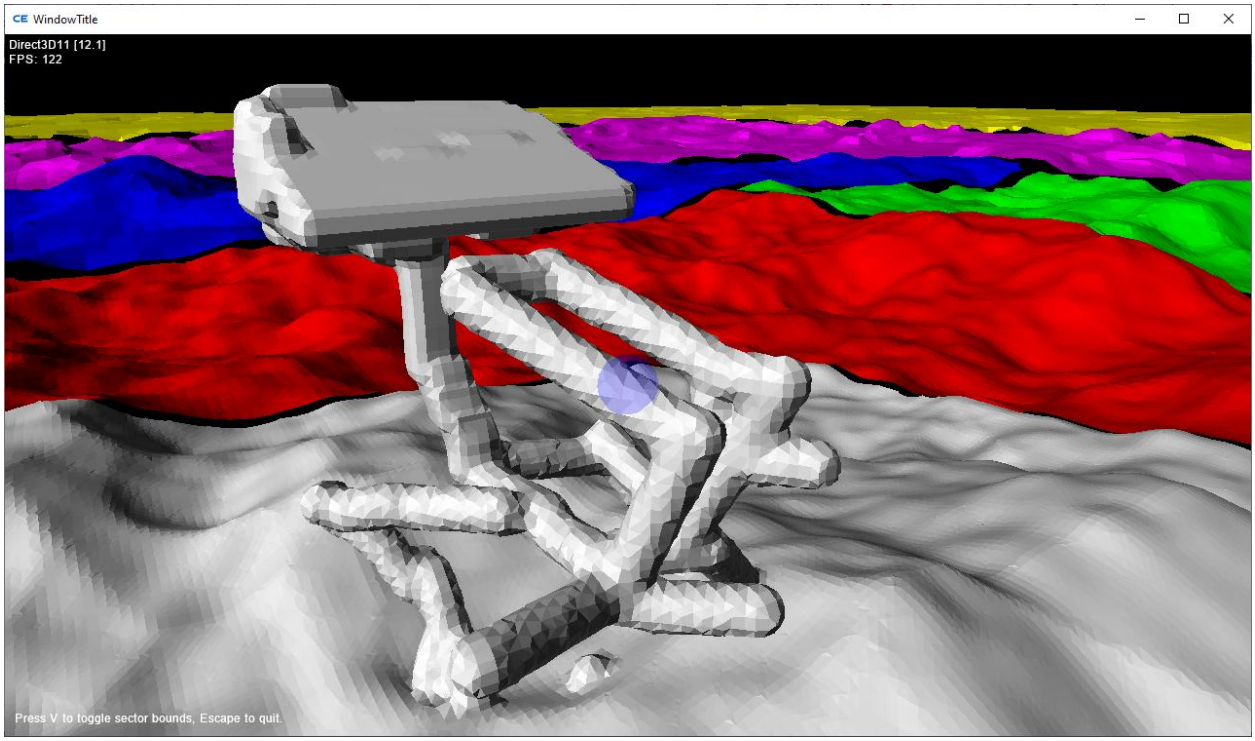


Figure 56 Voxel surface editing and rendering example with differing LODs.

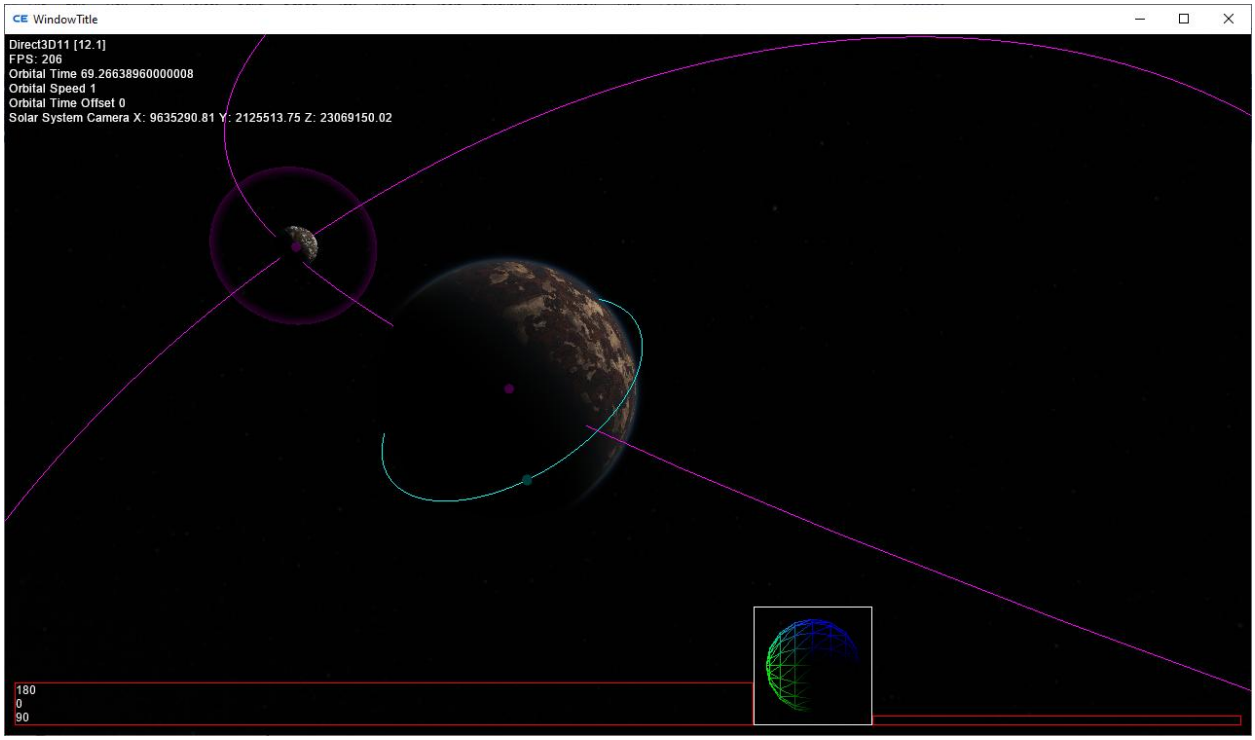


Figure 57 Example of planets and orbital lines within the orbital mechanics simulator.