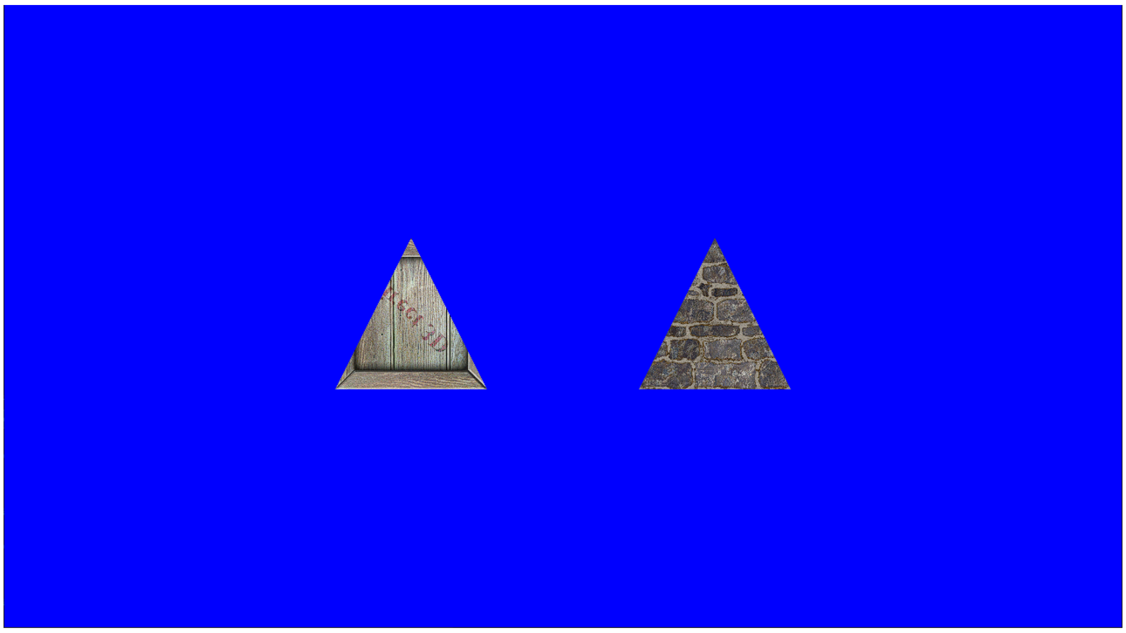
My First Ray Tracer

In The Skateboard Engine

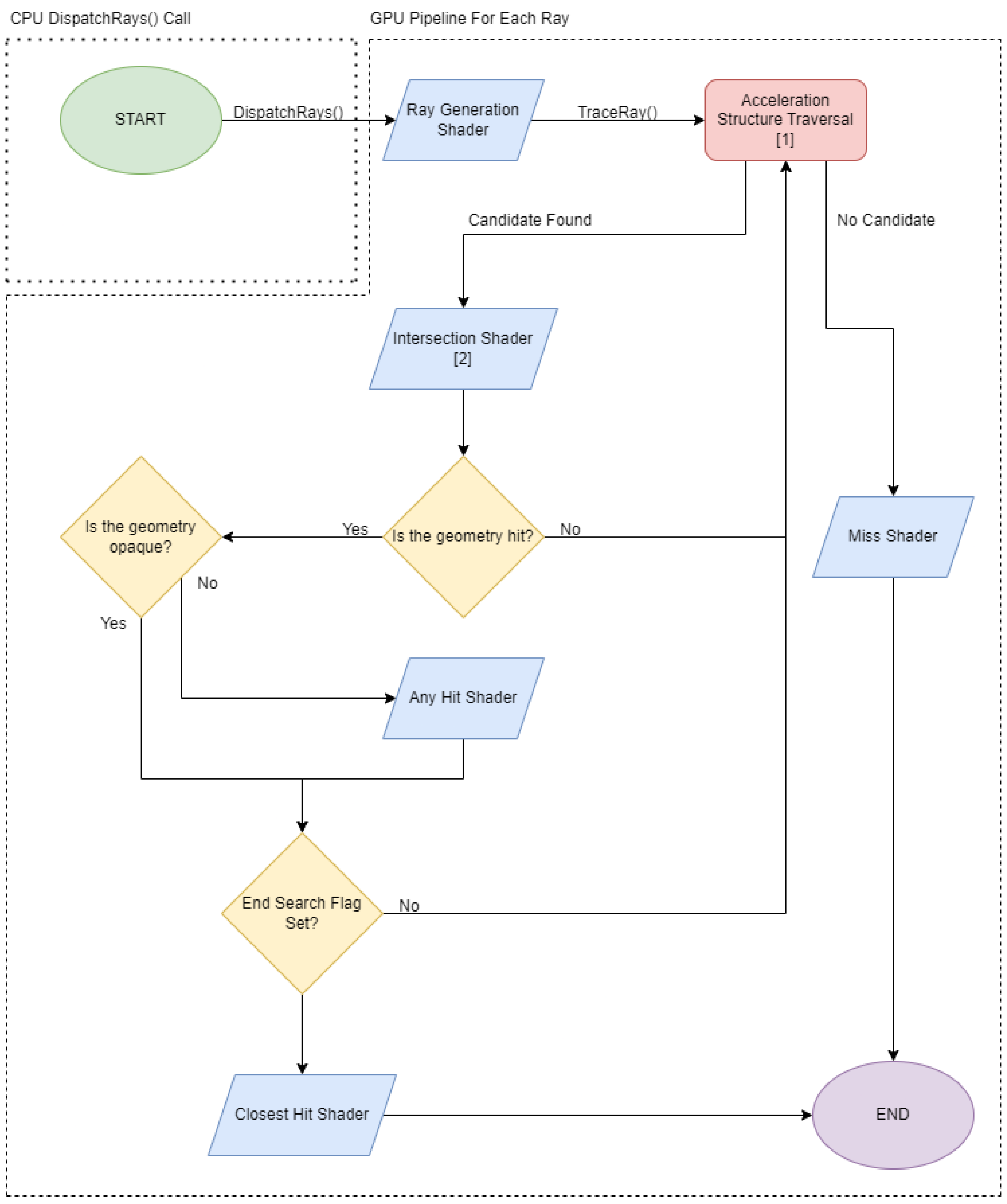
The purpose of this document is to give you an overview of the ray tracing technology and how it can be used in the *Skateboard* engine. In this document we will be assuming that you are comfortable with graphics concepts and with using this engine to create your buffers, textures, etc. You should also be reminded that ray tracing and mesh shading technologies require compatible hardware and will not be available to use in the engine otherwise. There are methods from the different APIs to emulate ray tracing with traditional compute shaders, but we have elected not to use these as it would ultimately lose the benefits of this pipeline in comparison to traditional rasterization. The topics covered in this document are brief and only serve as an introduction to ray tracing. If you are interested in going further into the subject, have a look at our *GameLayer Demo*, the DirectX Raytracing reference as well as the various documents presented in the Playstation devnet.
We can break our tasks down to the following list:

- Overview of the GPU accelerated Ray Tracing pipeline
- Creating bottom and top level acceleration structures for our geometry
- Creating our ray tracing shader library
- Creating the pipeline, dispatching our rays and outputting the result

It is noted that we will not cover tracing analytical geometry in this document, as this is only an introduction. However, we have developed a sample (*GameLayer Demo*) that demonstrates all the possibilities, including analytical shapes and ray traced shadows, for ray tracing in the *Skateboard* engine. Feel free to ask a tutor about this sample for more complex ray tracing rendering. You can find the full code that resulted from creating this document as appendices at the end.



GPU Accelerated Ray Tracing Pipeline Overview



[1] Enumerates candidates that potentially intersect the ray from the acceleration structure (BVH).
[2] This shader is only being called for user-defined primitive geometry. A built-in efficient ray-triangle intersection shader is used otherwise. For simplicity, this diagram shows that in either case an intersection shader is being called to check geometry intersection.

Figure 1 - Ray Tracing Pipeline Overview

Figure 1 shows you a basic overview of the ray tracing pipeline in DirectX Raytracing. Note that the programmable parts of this pipeline, i.e. the shaders, are displayed in blue. All the rest is handled by the API. This overview is only a simplification, refer to the full reference.

Top & Bottom Level Acceleration Structures

Unlinke rasterization, a ray tracer will require knowledge of the entire scene when rendering. In fact, we may want to have some rays bounce around the environment and gather the global illumination. Global illumination is considering the global transport of lighting as we observe it in the real world, that is light rays originating from different light sources and interacting with the environment until it has either lost its energy or reached an observer like our eyes. This effect is very difficult to obtain in rasterization since we draw objects one by one, with local data (i.e. local illumination). However, with ray tracing technologies, we could approach path tracing, a complete path of rays simulating the global light transportation. Therefore, it is impossible to consider tracing rays with local objects, and we will need a global description of our scene traceable at all times. Nevertheless, a problem arises when trying to identify intersections with all of these different pieces of geometry. In other words, how can you determine that a ray has hit an object? You could try the brute force way, and check with each geometry one by one, but you will quickly find out that your performance becomes unreasonably slow for real-time graphics. A better approach will be to use some sort of acceleration structure, a method to quickly identify what objects are more likely to intersect a given ray. This is where Top & Bottom Level Acceleration Structures (respectively TLAS and BLAS) come in, and luckily for us, the API will provide us with a way to very easily build them! At the bottom line, the choice for your acceleration structure will be made by a collaboration of a given hardware manufacturer and the developers of the API you are using. In DirectX Raytracing and PlayStation Raytracing, building an acceleration structure will result in a Bounding Volume Hierarchy (BVH). Using BVHs significantly improves the tracing process to the point that it can be performed in real-time, even for complex scenes with thousands of different pieces of geometry with reasonable complexity. When dealing with ray tracing in the *Skateboard* engine, we will therefore use these APIs to create our acceleration structures for our geometry, such that it can be used in our different ray tracers.
The first concept we will need to understand is the difference between a Top Level Acceleration Structure and a Bottom Level Acceleration Structure. This is described in extensive detail in the DirectX Raytracing reference, but we'll give you a brief overview here as well. For a given ray. your API will try to enumerate potential candidates for intersection, and, once a candidate has been elected as suitable, it will trace that particular object to obtain a hit. As seen in the overview in the previous section, tracing is resumed by a set of steps:

1. Enumerate and find a candidate
2. Trace the candidate
    a. If a hit was detected, register it as the current closest intersection
3. Continue enumeration to find a closer candidate until no more are found
4. Terminate tracing

Essentially, the enumeration will be performed on the Top Level Acceleration Structure. The TLAS is the global representation of your scene. In other words, it contains one or multiple BLAS for tracing. You can also note that you can make use of instancing as well, by referencing the same BLAS in multiple different locations. In contrast, you can think of a Bottom Level Acceleration Structure to optimize the tracing of a single object. While this is not necessarily true for AAA engines, it is the case for our implementations in the *Skateboard* engine. The BLAS will divide your mesh into several sections for faster tracing.

This can be visible using hardware developer tools, such as AMD's *RadeonRaytracingAnalyzer*.
Before continuing this section by walking you through the creation of acceleration structures, you should note that if you have used the engine's *SceneBuilder* it will have already created these acceleration structures for you if your computer was found compatible. You can always take our implementation in this builder as an example for your own geometry generators. Otherwise, follow along and we'll get started! Our goal in this section is to demonstrate how to create an acceleration structure for a single triangle.
Now, let's get started with the creation of the BLAS for our triangle. We will assume here that you have already constructed a vertex and index buffer for your geometry. The first thing we'll need to do, as for most of the *Skateboard* engine objects, is to fill in a *Skateboard::BottomLevelAccelerationStructureDesc* structure:

```
Skateboard::BottomLevelAccelerationStructureDesc blasDesc = {};
blasDesc.Type = Skateboard::GeometryType_Triangles;
blasDesc.Flags = Skateboard::GeometryFlags_Opaque;
blasDesc.Triangles.pVertexBuffer = p_VertexBuffer.get();
blasDesc.Triangles.StartVertexLocation = 0;
blasDesc.Triangles.VertexCount = vertexCount;
blasDesc.Triangles.pIndexBuffer = p_IndexBuffer.get();
blasDesc.Triangles.StartIndexLocation = 0;
blasDesc.Triangles.IndexCount = indexCount;
```

Note that since we are dealing with triangles for our mesh, we are filling in the *Triangles* section of the description after making sure to specify that the geometry type is indeed for triangles. Also note that we are flagging this geometry as *Opaque*, which means that no *AnyHit* shader will be executed for our geometry. We supply pointers to our vertex and index buffers, as well as their respective counts. If we were working with buffers containing data for multiple geometries, we could have used the *StartVertexLocation* and *StartIndexLocation* to tell our engine the start of our data of interest within these buffers (see the *SceneBuilder* implementation). Once our description is nicely filled up, creating a BLAS in the *Skateboard* engine will be as simple as the following:

```
v_BLAS.emplace_back(Skateboard::BottomLevelAccelerationStructure::Create(L"Triangle
BLAS", blasDesc));
```

..where *v_BLAS* is a vector of unique pointers (we want to have a vector since it would be tedious to store them separately) to *Skateboard::BottomLevelAccelerationStructure* objects. And that's all there is to it for us! The engine will handle the information internally and use the DirectX Raytracing API to construct the BLAS on the GPU for us.
Our next step is therefore to construct the global scene using the TLAS. For this exercise, we will demonstrate instancing with two instances of our previously built BLAS. Once again, we will need to fill in a *Skateboard::TopLevelAccelerationStructureDesc* structure. This is a bit trickier. Keep in mind that this has been specifically designed to work with our *SceneBuilder*, which supports instancing:

```
        // Setup our data for the instances
        std::vector<float4x4> transforms = { glm::translate(float3(-2.f, 0.f, 0.f)),
glm::translate(float3(2.f, 0.f, 0.f)) };
```

```
        std::vector<uint32_t> meshIDs = { 0u, 0u };
        std::vector<uint32_t> instanceIndices = { 0u, 1u };

        // Create the TLAS with two instances of our BLAS
        Skateboard::TopLevelAccelerationStructureDesc tlasDesc = {};
        tlasDesc.vBLAS = &v_BLAS;
        tlasDesc.vTransforms = std::move(transforms);
        tlasDesc.vMeshIDs = std::move(meshIDs);
        tlasDesc.vInstanceIndices = std::move(instanceIndices);
        p_TLAS.reset(Skateboard::TopLevelAccelerationStructure::Create(L"Top Level
Acceleration Structure", tlasDesc));
```

The first thing we notice is that the TLAS is indeed able to reference multiple instances by using the *vInstanceIndices* field of the description. Essentially, you would add as many zero-based unique indices in this container, whose size determines the total count of instances present in this TLAS. Likewise, each instance will need a transform, so we are providing a collection of them in *vTransforms*. The field *vMeshIDs* will be used to index the BLAS vector we created earlier. Since we have only one triangle mesh, both IDs are to be zero. We also note that the BLAS vector is inputted into the description via a pointer to avoid pointless copying or losing ownership of our vector. And just like that, we have successfully created our TLAS!

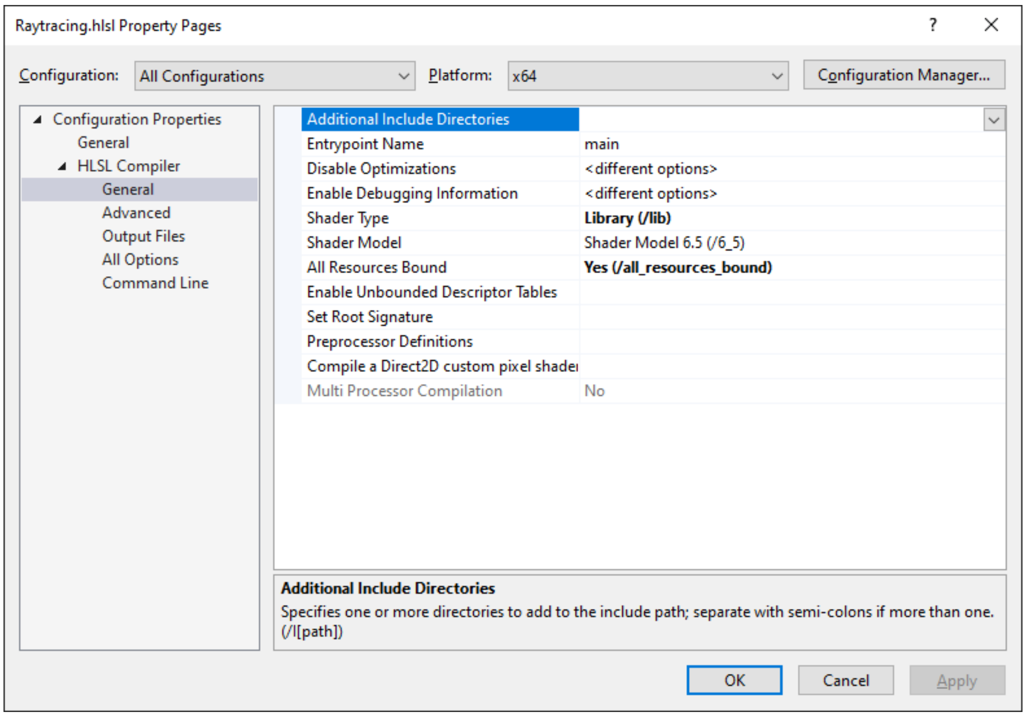
Before moving on to the next section, we will cover one last aspect of interest regarding TLASs: performing an update of the transform of an instance. Let’s say you modified the position of your geometry through the user interface, you will want to see these changes to be effective in your ray tracer. This is done by updating the TLAS through the graphics API. Typically, Top Level Acceleration Structures can be flagged for a fast trace (with slow build) or fast build (with slower trace). Since our engine doesn’t account for animations, which is where fast building would be interesting, the *Skateboard* TLASs will always be flagged to prefer fast tracing. In that sense, you should be careful not to update your TLAS every frame but rather on modification only. Performing an update is as simple as calling:

```
p_TLAS->PerformUpdate(<instanceIndex>, <newTransform>);
```

..where the *instanceIndex* is the index of the given instance you wish to modify from all the existing instances in this TLAS, and the *newTransform* is a *float4x4* transform that will replace the previous one for this instance. Note that if you construct your own instancing system, *instanceIndex* is used to index your instance buffer. That is, instance indices in the *Skateboard* engine are always unique and you will never have an instance of two separate meshes being the same index. Otherwise you will end up with updating two mesh instances to the same transform. In other words, if I constructed a second BLAS and wanted to assign two new instances of it in the TLAS, they would be *{ 2, 3 }* since my triangle instances already occupied indices 0 & 1. These indices are not to be confused with instance IDs, which are used for rasterization exclusively.

Creating the Ray Tracing Shader Library

As you have seen from the pipeline overview, the stages we can consider are the *Ray Generation Shader*, *Intersection Shader*, *Any Hit Shader*, *Closest Hit Shader* and the *Miss Shader*. However, only the *Ray Generation Shader* is mandatory, as it is executed directly after your dispatch. Not using any other shader just means you will obtain the *Closest T* from your tracing, a value that indicates how far from the origin of the ray the hit was detected. As this is an introduction to ray tracing, we will consider a minimalistic approach with a *Ray Generation Shader*, a *Closest Hit Shader* and a *Miss Shader*. That way, we will be able texture our triangle on hit or output a background colour on miss.
A good thing to know is that you can recursively trace your rays from certain shader stages. There are limitations to this, one of which we will have to define during the pipeline creation. But we won't be doing that here, so you should refer once again to our *GameLayer Demo* or the official documentation for more details on where and how that is allowed. Even though that is a possibility, some engineers prefer not to use recursion and rather only trace rays from the *Ray Generation Shader*. This is the pattern we will use. The idea is that our *Closest Hit Shader* will fill in a payload with information that we can use to shade our pixel back in the *Ray Generation Shader*. Using the recursive pattern, shading would occur on hit. Our pattern avoids the problems of cache allocation for different recursion levels. However, both patterns are, realistically, equally efficient from my own experience.
Let's get started and add a shader library to your *GameApp* project (right-click on *src > Add > New Item… > in the name field type Raytracing.hlsl*). We will have to configure this HLSL as a library, so let's head to its properties (right-click on *Raytracing.hlsl* on your *Solution Explorer > Properties > Configuration Properties > HLSL Compiler > General*) and make sure that the *Shader Type* is set to ***Library (/lib)***, the *Shader Model* is set to ***Shader Model 6.5 (/6_5)*** or above, and that *All Resources Bound* is set to ***yes (/all_resources_bound)***.


Raytracing.hlsl Property Pages

Configuration: All Configurations Platform: x64 Configuration Manager...

Additional Include Directories	
Entrypoint Name	main
Disable Optimizations	<different options>
Enable Debugging Information	<different options>
Shader Type	**Library (/lib)**
Shader Model	Shader Model 6.5 (/6_5)
All Resources Bound	**Yes (/all_resources_bound)**
Enable Unbounded Descriptor Tables	
Set Root Signature	
Preprocessor Definitions	
Compile a Direct2D custom pixel shade	
Multi Processor Compilation	No

Additional Include Directories
Specifies one or more directories to add to the include path; separate with semi-colons if more than one. (/I[path])

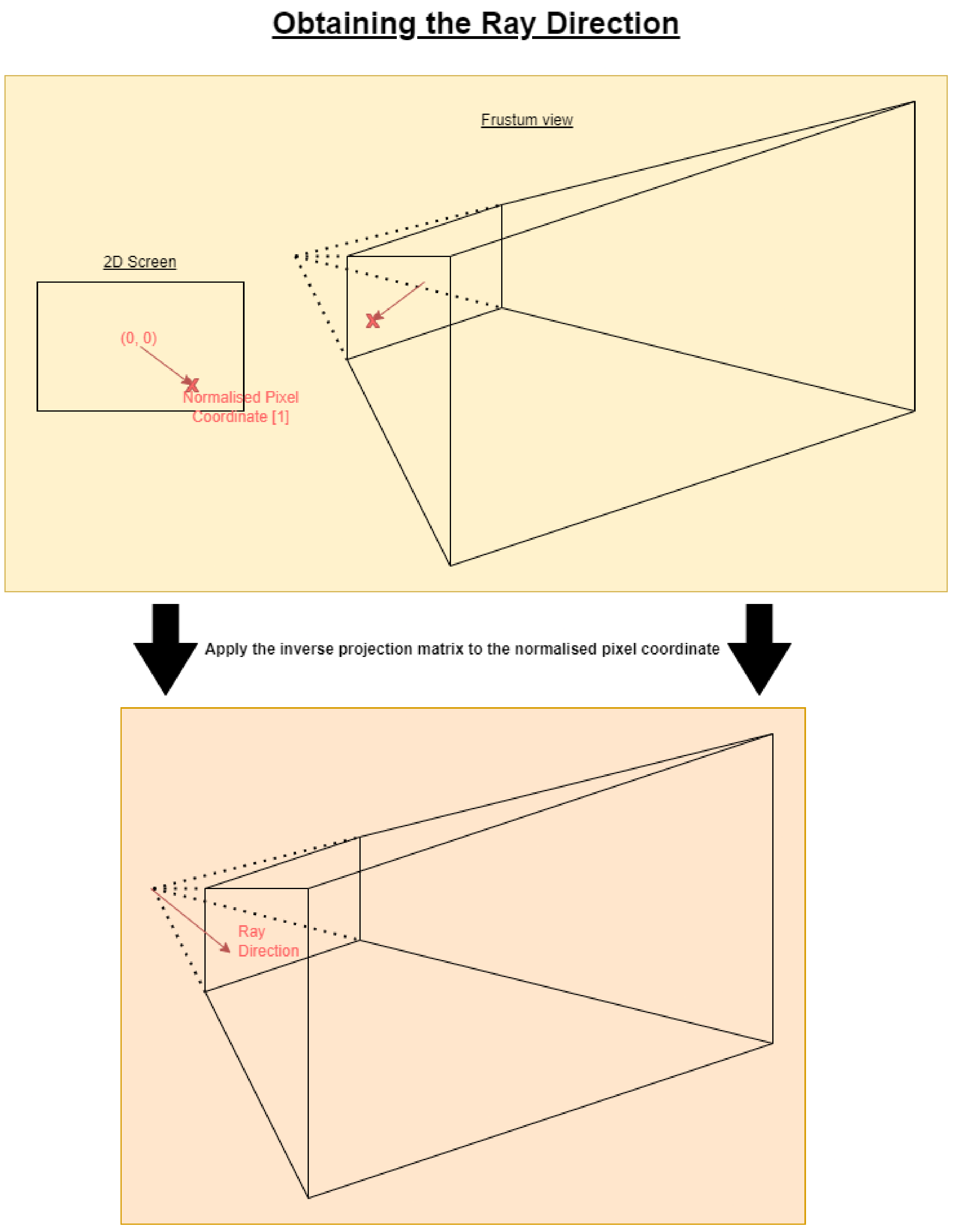
Figure 2 - Configuration Properties of the Ray Tracing Shader Library

Inside *Raytracing.hlsl*, we will start by adding the entry points of our shaders:

```
struct RadiancePayload
{
};

[shader("raygeneration")]
void RayGenShader()
{
}

[shader("closesthit")]
void ClosestHitShader(inout RadiancePayload payload, in
BuiltInTriangleIntersectionAttributes attributes)
{
}

[shader("miss")]
void MissShader(inout RadiancePayload payload)
{
}
```

You can immediately note that our shaders are being identified with a ***type attribute*** on top of the custom function name. Here I have created generic names, but you could have named your *Ray Generation Shader* something like *void ABCShader()* and considering you have added the correct *type attribute* on top of it the compiler will have no problem with that. You can also observe that both the closest hit and miss shaders expect a payload. This payload is how we will transfer information between our shader stages. According to the documentation, **it is very important to keep this payload with as little data as possible**. In this snippet the payload is empty, but we will be adding things into it shortly. The last observation we can make is the *BuiltInTriangleIntersectionAttributes* argument in the *Closest Hit Shader*. This is the result obtained from the intersection shader, so when dealing with triangles we can use the built-in attributes, which are barycentric coordinates that we will use to find our vertex. When you have your own analytical shapes with a custom intersection shader, you would add your own attribute structure here just as we did with the payload.

Before we can trace our rays, we will need to generate them from the *Ray Generation Shader* based on our camera view and projection matrices. In contrast to rasterization, we won't be using the standard view and projection matrices, but rather their inverse, as we are unable to transform geometries from their acceleration structures. In other words, instead of switching the world to the origin and projecting, we will cast our rays from the camera's world location and view, and use a deprojection to transform our pixel coordinates to proper 3D rays. See *Figure 3*.


Obtaining the Ray Direction
Frustum view
2D Screen
(0, 0)
Normalised Pixel Coordinate [1]
Apply the inverse projection matrix to the normalised pixel coordinate
Ray Direction

[1] A normalised pixel coordinate is in the range [-1,1]. Essentially, the origin (0,0) is being shifted from the top left to the center of the screen in order to deproject the origin-to-pixel vector into a ray direction about the world origin. It is noted that if a camera view is considered, a follow up multiplication by the inverse view matrix is needed to ensure the ray is casted in the right direction in world space.

Figure 3 - Generating the ray in 3D space from the screen coordinates

We will therefore consider a more elaborate *PassBuffer* as found in the snippet below. We will also need to define our *VertexType*, that is a structure representing the layout of our vertices in the vertex buffer. Unlike rasterization, the vertex buffer is not bound to the pipeline directly, but rather we will have to supply it as a shader resource view. That is valid for the index buffer as well. If you are working with a large vertex buffer, containing multiple pieces of geometry in it, you will have to consider an extra SRV to supply the offsets where each geometry starts. Additionally, we will need to reference the output UAV, that is a texture to which we will write the result of our tracing. This UAV is part of the *Skateboard::RaytracingPipeline* class and will be provided on register (*u0, space1*). Finally, we also need to reference the top level acceleration structure we have built earlier and the textures that we will supply on the pipeline description:

```
struct VertexType
{
	float3 position;
	float2 texCoord;
};

struct PassBuffer
{
	float4x4 ViewMatrix;
	float4x4 ProjectionMatrix;
	float4x4 ViewMatrixInverse;
	float4x4 ProjectionMatrixInverse;
};

RWTexture2D<float4> gOutput : register(u0, space1);
RaytracingAccelerationStructure gSceneBVH : register(t0, space0);
ConstantBuffer<PassBuffer> gPassBuffer : register(b0, space0);
StructuredBuffer<VertexType> gVertices : register(t1, space0);
StructuredBuffer<uint> gIndices : register(t2, space0);
//StructuredBuffer<uint> gVertexOffsets : register(t3, space0);
//StructuredBuffer<uint> gIndexOffsets : register(t4, space0);
Texture2D<float4> gTextures[] : register(t0, space2);
SamplerState gSampler : register(s0, space0);
```

Now, since this is an introduction, and this document is already getting quite large, we'll consider the helper functions below for speeding up the process. I won't be going into details as I've added sensible comments to help you understand how they work. But essentially, you have a function that will generate a ray from our camera inverse matrices as explained and illustrated above in *Figure 3*, and a function that will use the barycentric coordinates retrieved from the closest hit result to interpolate between our triangle vertices:

```
void GenerateCameraRay(out float3 origin, out float3 direction)
{
	// Get the location within the dispatched 2D grid of work items
	// (often maps to pixels, so this could represent a pixel coordinate).
	uint2 launchIndex = DispatchRaysIndex().xy;
	float2 dims = float2(DispatchRaysDimensions().xy);

	// Calculated the floating point pixel coordinate normalised on [0,1] x [0,1]
	float2 pixelCoordNormalised = (((launchIndex.xy + 0.5f) / dims.xy) * 2.f - 1.f);

	// Perspective ray (see hand written notes)
	// Note that the matrices are transposed since the Nvidia camera buffer did not
transpose them on the CPU
	// This could be avoided by pre-multiplying (i.e. mul(viewI, float4(0,0,0,1)); etc.)
	origin = mul(gPassBuffer.ViewMatrixInverse, float4(0, 0, 0, 1)).xyz;
									// The
view matrix dictates the origin of the ray. Retrieve the origin by mutiplying the inverse view
with the world origin (0, 0, 0)
	float4 target = mul(gPassBuffer.ProjectionMatrixInverse,
float4(pixelCoordNormalised.x, -pixelCoordNormalised.y, 1, 1));	// Apply the inverse
projection to the pixel coordinate. This creates a target vector from (0,0,0) to pixCoord
	direction = mul(gPassBuffer.ViewMatrixInverse, float4(target.xyz, 0)).xyz;
									// The
direction is origin -> pixel coordinate
}

VertexType GetInterpolatedVertex(/*in const uint meshID,*/ in const uint primitiveIndex, in
const float2 bary)
{
	// Calculate gamma
	// Barycentric coordinates: alpha + beta + gamma = 1
	// Thus: gamma = 1 - alpha - beta
	const float alpha = bary.y;
	const float beta = bary.x;
	const float gamma = 1.f - alpha - beta;
	const float3 barycentrics = float3(gamma, beta, alpha);

	// Plan
	// 1) Get the base vertex buffer index and the base index buffer index
	// 2) Retrieve the current primitive indices (3 indices per triangle)
	// 3) Find the 3 vertices of the primitive using the indices and the base vertex buffer
index
	//		-> vertexBufferPos[i] = baseVertexPos + index[i]
	// 4) Apply the barycentric weighting on each vertex and add them all together
	//		-> This will essentially interpolate between them
	//		-> gamma * vertex[0] + beta * vertex[1] + alpha * vertex[2]

	// 1
	// Here we have 0u as our vertex and index buffers only describe the triangle
	// If you are packing your geometry into the same buffers, then you would
	// need to consider their offsets instead
	const uint baseVertexLocation = /*gVertexOffsets[meshID]*/ 0u;
	const uint baseIndexLocation = /*gIndexOffsets[meshID]*/ 0u;

	// 2
	const uint baseIndexOffset = 3u * primitiveIndex;
	const uint3 primitiveIndices = uint3(
		gIndices[baseIndexLocation + baseIndexOffset + 0u],
		gIndices[baseIndexLocation + baseIndexOffset + 1u],
		gIndices[baseIndexLocation + baseIndexOffset + 2u]
	);

	VertexType vertex;
	[unroll] for (uint i = 0; i < 3u; i++)
	{
```

```
                // 3
                const uint currentVertexLocation = baseVertexLocation + primitiveIndices[i];
                VertexType temp = gVertices[currentVertexLocation];

                // 4
                vertex.position += barycentrics[i] * temp.position;
                vertex.texCoord += barycentrics[i] * temp.texCoord;

                // Other attributes you may supply
                //vertex.normal         += barycentrics[i] * temp.normal;
                //vertex.tangent        += barycentrics[i] * temp.tangent;
                //vertex.bitangent      += barycentrics[i] * temp.bitangent;
        }
        //vertex.normal = normalize(vertex.normal);
        //vertex.tangent = normalize(vertex.tangent);
        //vertex.bitangent = normalize(vertex.bitangent);

        return vertex;
}
```

Okay, that was a lot of material to take in already. But with that, we are finally able to start writing our shaders. The first thing we will consider is to generate and trace our ray from the *Ray Generation Shader*:

```
// Generate the viewport ray
float3 rayOrigin, rayDirection;
GenerateCameraRay(rayOrigin, rayDirection);

// Trace the ray
RayDesc ray;
RadiancePayload rayPayload = (RadiancePayload)0;
ray.Origin = rayOrigin;
ray.Direction = rayDirection;
ray.TMin = .1f;
ray.TMax = 100000.f;
TraceRay(
        gSceneBVH,
        RAY_FLAG_NONE,
        0xFF,
        0,
        0,
        0,
        ray,
        rayPayload
);

// Calculate a shading colour from the payload information
float4 output = float4(0.f, 0.f, 0.f, 0.f);

// Write to the output
gOutput[DispatchRaysIndex().xy] = output;
```

The idea is that we first generate a ray based on our camera inverse matrices as explained earlier. This ray is originating at the camera position and in the correct direction. We then need to tell the GPU to start tracing our ray using the TraceRay() function. Details are explained in the official documentation, but for now these parameters will be a good default basis. Note that the *RayDesc* object is built into HLSL, we do not have to define it. The *TMin* value is non-zero to ensure that a ray will not intersect a geometry when bouncing off of a surface. The *TMax* value defines how far you wish to trace, but it does not matter if you input a small or large maximum distance (it will just reject candidates found to be further than the current *closestT*, which is set to *TMax* at the start), so we will just make sure to trace everything we can with a large number.

We now need to start making use of the payload. The easiest way for us to handle our payload is just to transfer the interpolated vertex texture coordinates we will obtain in the closest shader later on, the instance ID, and to flag whether a hit occurred or not. Recall that the payload size needs to be minimised for maximum performance. If your payload size starts to become very large, then using the recursion pattern and shading inside the *Closest Hit Shader* might be preferable. Otherwise, for small payloads like ours, we will shade in the *Ray Generation Shader* to minimise cache consumption as a consequence of recursion. Therefore, our payload simply becomes:

```
struct RadiancePayload
{
	float2 uv;
	uint instanceID;
	bool hit;
};
```

We can then use this information to sample our texture when a ray was hit. When the ray misses, we will return a bright blue colour to contrast the back-buffer’s default dark grey, so we will easily know that our miss shader did work. To determine which texture to sample, we will use the instance ID we had inputted on the top level acceleration structure description before its construction. There is a little catch, however, when it comes to texture sampling: we need to use the intrinsic ***NonUniformResourceIndex()*** when indexing arrays of resources like our texture array. I won’t go too deep into the details here, but essentially not every thread of your GPU will obtain the same index within a thread wave, and this behaviour is therefore non-uniform. Nvidia GPUs will automatically handle this, so you may find that it has no effect on your machine, but AMD GPUs will need it (at least at the time of writing this document they still do). Our colouring code therefore becomes:

```
// Calculate a shading colour from the payload information
float4 output = float4(0.f, 0.f, 1.f, 1.f);
	if(rayPayload.hit)
		output =
gTextures[NonUniformResourceIndex(rayPayload.instanceID)].SampleLevel(gSampler,
rayPayload.uv, 0);
```

Note that we had to use *SampleLevel* to sample our texture, as we are effectively not in a pixel shader, but rather a compute-like environment.

We can now define what happens when the ray has hit or missed our geometry. We will start by handling the behaviour when the ray misses and simply return that no hit has occurred:

```
payload.hit = false;
```

Finally, we will define what happens when the ray hits a piece of geometry. We will make use of the *GetInterpolatedVertex* helper function to retrieve our vertex data, and supply the texture coordinate to our payload. Note that we need to use the *PrimitiveIndex()* intrinsic to identify which triangle in our mesh we are intersecting. Since our mesh consists of a single triangle, this will always be zero, but for more complicated meshes with many triangles you will use this intrinsic to identify the triangle for interpolation. The intrinsic *InstanceID()* is the ID we have inputted on our TLAS construction. We can only retrieve it within the *TraceRay()* scope, so we have to pass it to our payload so we can use it to sample the texture. Our *ClosestHitShader* therefore becomes:

```
// Retrieve the interpolated vertex on our triangle
VertexType vertex = GetInterpolatedVertex(PrimitiveIndex(), attributes.barycentrics);

// Assign the payload
payload.uv = vertex.texCoord;
payload.instanceID = InstanceID();
payload.hit = true;
```

And that's it! That was a lot of work and material to take in, good job for making it this far! There is a lot more to cover and we did not use all the shader stages, but this will be a good start for your first steps into ray tracing! The only thing left to do is to go back to our *RaytracingLayer.cpp* and create the pipeline.

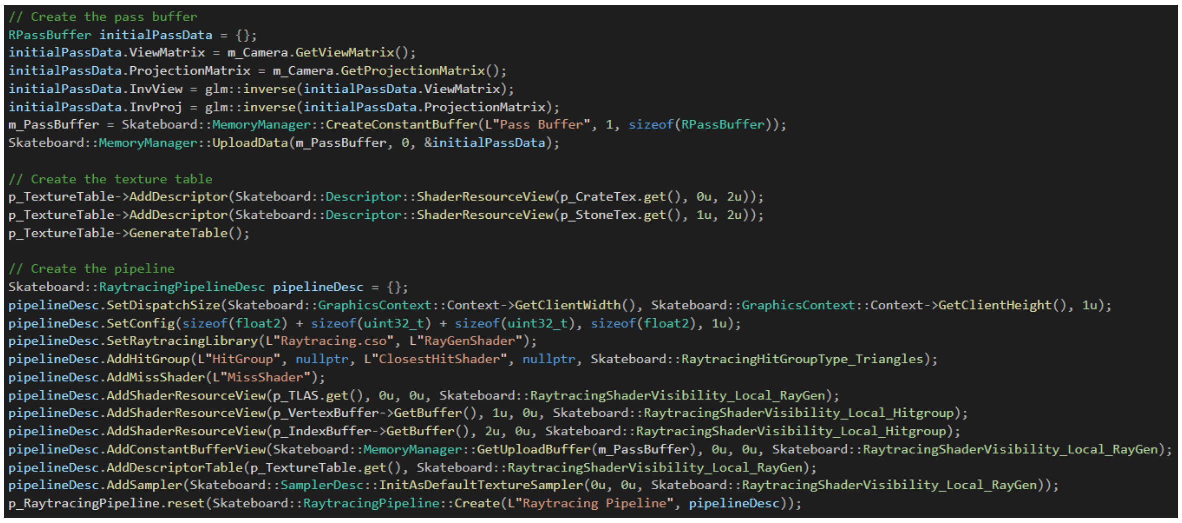
Creating the Ray Tracing Pipeline and Dispatch

With acceleration structures built and ready to be used, we can tackle the creation of our ray tracing pipeline. This does not differ much from the rasterization pipeline creation; that is we must create a *Skateboard::RaytracingPipelineDesc* and fill it up with our buffers, shaders and textures. *Figure 4* shows how we implemented the pipeline with our constant buffer with the camera information. Note how we obtain the inverse of our view and projection matrices using *glm::inverse()*. The *SetConfig()* function is important as this is where you configure the **maximum size** of your **payload** and **attributes**. Since our payload consisted of *{ float2, uint, bool }* in the shader, the maximum size is 16 bytes (2 * 4 + 4 + 4, recall that in HLSL a *bool* is 4 bytes large). As for our attributes, we only used the default triangle attributes which give 2 floats, and therefore a size of 8 bytes. An important thing to note here as well is that you can make your resources to certain stages only. There are four possible options:

1. Global
2. Local to the Ray Generation Shader
3. Local to the HitGroup (Intersection Shader, Any Hit Shader, Closest Hit Shader)
4. Local to the Miss Shader

Making resources visible to certain stages only can improve the performance of the program, so in this example we have made our resources visible to the stages where they would be used only.

A little note on HitGroups: a HitGroup regroups all the shaders involved in determining a hit. You can have multiple HitGroups for different kinds of geometries or purposes. For example, our more complex *GameLayer Demo*, the ray tracing implementation with the *SceneBuilder* generates 4 groups: one for normal shading, one transparent objects with shadows, one for analytical shading, one for transparent analytical shadows. Although transparency isn't supported yet, they have been added as examples on how they could be implemented.



```cpp
// Create the pass buffer
RPassBuffer initialPassData = {};
initialPassData.ViewMatrix = m_Camera.GetViewMatrix();
initialPassData.ProjectionMatrix = m_Camera.GetProjectionMatrix();
initialPassData.InvView = glm::inverse(initialPassData.ViewMatrix);
initialPassData.InvProj = glm::inverse(initialPassData.ProjectionMatrix);
m_PassBuffer = Skateboard::MemoryManager::CreateConstantBuffer(L"Pass Buffer", 1, sizeof(RPassBuffer));
Skateboard::MemoryManager::UploadData(m_PassBuffer, 0, &initialPassData);

// Create the texture table
p_TextureTable->AddDescriptor(Skateboard::Descriptor::ShaderResourceView(p_CrateTex.get(), 0u, 2u));
p_TextureTable->AddDescriptor(Skateboard::Descriptor::ShaderResourceView(p_StoneTex.get(), 1u, 2u));
p_TextureTable->GenerateTable();

// Create the pipeline
Skateboard::RaytracingPipelineDesc pipelineDesc = {};
pipelineDesc.SetDispatchSize(Skateboard::GraphicsContext::Context->GetClientWidth(), Skateboard::GraphicsContext::Context->GetClientHeight(), 1u);
pipelineDesc.SetConfig(sizeof(float2) + sizeof(uint32_t) + sizeof(uint32_t), sizeof(float2), 1u);
pipelineDesc.SetRaytracingLibrary(L"Raytracing.cso", L"RayGenShader");
pipelineDesc.AddHitGroup(L"HitGroup", nullptr, L"ClosestHitShader", nullptr, Skateboard::RaytracingHitGroupType_Triangles);
pipelineDesc.AddMissShader(L"MissShader");
pipelineDesc.AddShaderResourceView(p_TLAS.get(), 0u, 0u, Skateboard::RaytracingShaderVisibility_Local_RayGen);
pipelineDesc.AddShaderResourceView(p_VertexBuffer->GetBuffer(), 1u, 0u, Skateboard::RaytracingShaderVisibility_Local_Hitgroup);
pipelineDesc.AddShaderResourceView(p_IndexBuffer->GetBuffer(), 2u, 0u, Skateboard::RaytracingShaderVisibility_Local_Hitgroup);
pipelineDesc.AddConstantBufferView(Skateboard::MemoryManager::GetUploadBuffer(m_PassBuffer), 0u, 0u, Skateboard::RaytracingShaderVisibility_Local_RayGen);
pipelineDesc.AddDescriptorTable(p_TextureTable.get(), Skateboard::RaytracingShaderVisibility_Local_RayGen);
pipelineDesc.AddSampler(Skateboard::SamplerDesc::InitAsDefaultTextureSampler(0u, 0u, Skateboard::RaytracingShaderVisibility_Local_RayGen));
p_RaytracingPipeline.reset(Skateboard::RaytracingPipeline::Create(L"Raytracing Pipeline", pipelineDesc));
```

Figure 4 - Creation of the ray tracing pipeline

With that, the only thing left to do is to dispatch our rays and to copy the result to our back buffer (or, optionally, frame buffer). This is done during rendering, so let's head to the *OnRender()* function of your layer and add the following commands:

```
Skateboard::RenderCommand::DispatchRays(p_RaytracingPipeline.get());
Skateboard::RenderCommand::CopyUAVToBackBuffer(p_RaytracingPipeline->GetOutputUAV());
```

Now compile and run your program. If everything went right, you should see your two triangle instances rendering with the two different textures you have loaded! Congratulations for following through all of this, hopefully most of it made sense and don't get too frustrated if things are not working at first! This is a very difficult topic and may require quite a lot of headaches before you get anything on the screen (:

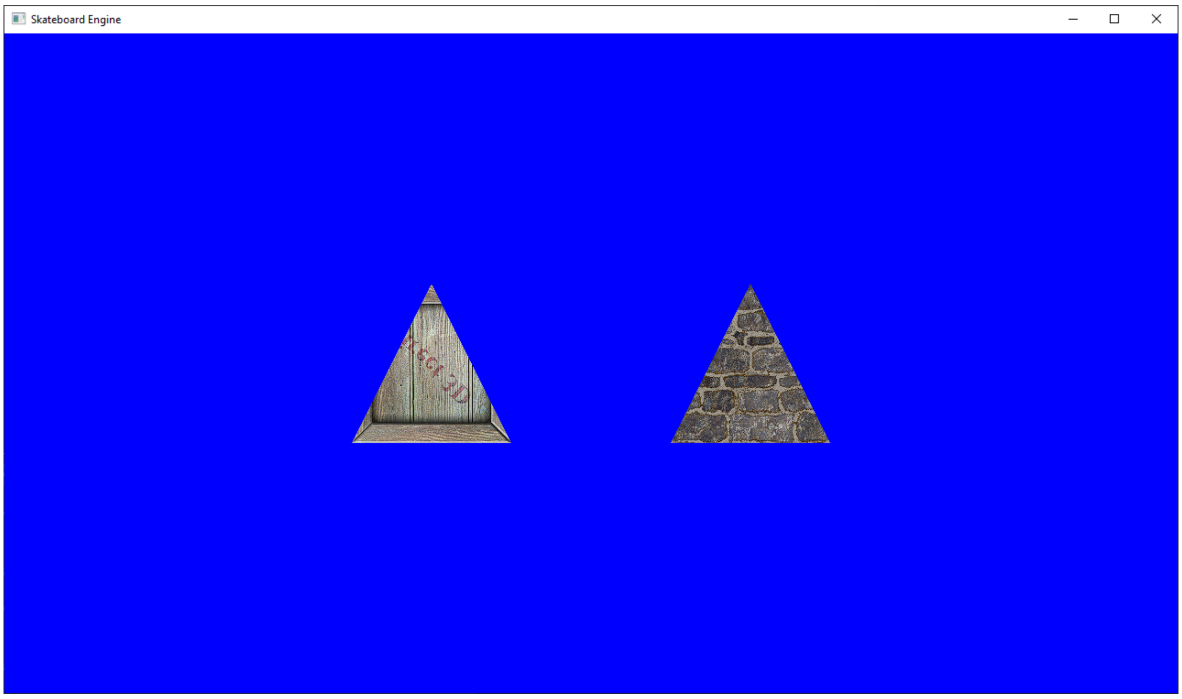


Figure 5 - Final output (yay!)

Appendix A - RaytracingLayer.h

```
#pragma once
#include <Skateboard.h>

struct RPassBuffer
{
	float4x4 ViewMatrix;
	float4x4 ProjectionMatrix;
	float4x4 InvView;
	float4x4 InvProj;
};

class RaytracingLayer final : public Skateboard::Layer
{
public:
	RaytracingLayer();
	virtual ~RaytracingLayer() final override;

	virtual void OnResize(int newClientWidth, int newClientHeight) final override;	//
Called when the platform called for a resize of the main application
	virtual void OnAttach() final override;
					// Called when this layer is being attached to the main
application (PushLayer)
	virtual void OnDetach() final override;
						// Called when this layer is being detached from the
main application (PopLayer)
	virtual bool OnHandleInput(float dt) final override;
		// Used to handle the inputs according to your game logic
	virtual bool OnUpdate(float dt) final override;
			// Used to update your scene and objects
	virtual void OnRender() final override;
						// Used to render your scene and objects
	virtual void OnImGuiRender() final override;
			// Used to render your user interface

private:
	std::unique_ptr<Skateboard::Texture> p_CrateTex;
	std::unique_ptr<Skateboard::Texture> p_StoneTex;
	std::unique_ptr<Skateboard::DescriptorTable> p_TextureTable;
	Skateboard::PerspectiveCamera m_Camera;
	uint32_t m_PassBuffer;

	std::unique_ptr<Skateboard::VertexBuffer> p_VertexBuffer;
	std::unique_ptr<Skateboard::IndexBuffer> p_IndexBuffer;
	std::vector<std::unique_ptr<Skateboard::BottomLevelAccelerationStructure>>
v_BLAS;
	std::unique_ptr<Skateboard::TopLevelAccelerationStructure> p_TLAS;
	std::unique_ptr<Skateboard::RaytracingPipeline> p_RaytracingPipeline;
};
```

Appendix B - RaytracingLayer.cpp

```
#include "RaytracingLayer.h"

RaytracingLayer::RaytracingLayer() :
	p_CrateTex(Skateboard::AssetManager::LoadTexture(L"assets/crate.dds")),
	p_StoneTex(Skateboard::AssetManager::LoadTexture(L"assets/stone.dds")),

p_TextureTable(Skateboard::DescriptorTable::Create(Skateboard::DescriptorTableDesc::In
it(Skateboard::ShaderDescriptorTableType_CBV_SRV_UAV))),
	m_Camera(.25f * SKTBD_PI,
Skateboard::GraphicsContext::Context->GetClientAspectRatio(), .1f, 1000.f, float3(0.f, 0.f,
-10.f), float3(0.f, 0.f, -9.f), float3(0.f, 1.f, 0.f)),
	m_PassBuffer(0u)
{
	// Create a vertex layout to define the vertex structure
	Skateboard::BufferLayout vertexLayout = {
		{ "POSITION", Skateboard::ShaderDataType_::Float3 },
		{ "TEXCOORD", Skateboard::ShaderDataType_::Float2 }
	};

	float vertices[] = {
		-1.f, -1.f, 0.f,		0.f, 0.f,
		0.f, 1.f, 0.f,			.5f, 1.f,
		1.f, -1.f, 0.f,			1.f, 0.f
	};
	const uint32_t vertexCount = sizeof(vertices) / vertexLayout.GetStride();
	p_VertexBuffer.reset(Skateboard::VertexBuffer::Create(L"Triangle Vertices",
vertices, vertexCount, vertexLayout));

	uint32_t indices[] = {
		0u, 1u, 2u
	};
	const uint32_t indexCount = _countof(indices);
	p_IndexBuffer.reset(Skateboard::IndexBuffer::Create(L"Triangle Indices", indices,
indexCount));

	// Create the BLAS
	Skateboard::BottomLevelAccelerationStructureDesc blasDesc = {};
	blasDesc.Type = Skateboard::GeometryType_Triangles;
	blasDesc.Flags = Skateboard::GeometryFlags_Opaque;
	blasDesc.Triangles.pVertexBuffer = p_VertexBuffer.get();
	blasDesc.Triangles.StartVertexLocation = 0;
	blasDesc.Triangles.VertexCount = vertexCount;
	blasDesc.Triangles.pIndexBuffer = p_IndexBuffer.get();
	blasDesc.Triangles.StartIndexLocation = 0;
	blasDesc.Triangles.IndexCount = indexCount;

v_BLAS.emplace_back(Skateboard::BottomLevelAccelerationStructure::Create(L"Triangle
BLAS", blasDesc));

	// Setup our data for the instances
	std::vector<float4x4> transforms = { glm::translate(float3(-2.f, 0.f, 0.f)),
glm::translate(float3(2.f, 0.f, 0.f)) };
	std::vector<uint32_t> meshIDs = { 0u, 0u };			// The same triangle will
give the same ID for both instances
	std::vector<uint32_t> instanceIndices = { 0u, 1u };

	// Create the TLAS with two instances of our BLAS
	Skateboard::TopLevelAccelerationStructureDesc tlasDesc = {};
	tlasDesc.vBLAS = &v_BLAS;
	tlasDesc.vTransforms = std::move(transforms);
	tlasDesc.vMeshIDs = std::move(meshIDs);
	tlasDesc.vInstanceIndices = std::move(instanceIndices);
	p_TLAS.reset(Skateboard::TopLevelAccelerationStructure::Create(L"Top Level
Acceleration Structure", tlasDesc));

	// Create the pass buffer
	RPassBuffer initialPassData = {};
	initialPassData.ViewMatrix = m_Camera.GetViewMatrix();
	initialPassData.ProjectionMatrix = m_Camera.GetProjectionMatrix();
	initialPassData.InvView = glm::inverse(initialPassData.ViewMatrix);
	initialPassData.InvProj = glm::inverse(initialPassData.ProjectionMatrix);
	m_PassBuffer = Skateboard::MemoryManager::CreateConstantBuffer(L"Pass
Buffer", 1, sizeof(RPassBuffer));
	Skateboard::MemoryManager::UploadData(m_PassBuffer, 0, &initialPassData);

	// Create the texture table

p_TextureTable->AddDescriptor(Skateboard::Descriptor::ShaderResourceView(p_CrateTe
x.get(), 0u, 2u));

p_TextureTable->AddDescriptor(Skateboard::Descriptor::ShaderResourceView(p_StoneTe
x.get(), 1u, 2u));
	p_TextureTable->GenerateTable();

	// Create the pipeline
	Skateboard::RaytracingPipelineDesc pipelineDesc = {};

pipelineDesc.SetDispatchSize(Skateboard::GraphicsContext::Context->GetClientWidth(),
Skateboard::GraphicsContext::Context->GetClientHeight(), 1u);
	pipelineDesc.SetConfig(sizeof(float2) + sizeof(uint32_t) + sizeof(uint32_t),
sizeof(float2), 1u);
	pipelineDesc.SetRaytracingLibrary(L"Raytracing.cso", L"RayGenShader");
	pipelineDesc.AddHitGroup(L"HitGroup", nullptr, L"ClosestHitShader", nullptr,
Skateboard::RaytracingHitGroupType_Triangles);
	pipelineDesc.AddMissShader(L"MissShader");
	pipelineDesc.AddShaderResourceView(p_TLAS.get(), 0u, 0u,
Skateboard::RaytracingShaderVisibility_Local_RayGen);
	pipelineDesc.AddShaderResourceView(p_VertexBuffer->GetBuffer(), 1u, 0u,
Skateboard::RaytracingShaderVisibility_Local_Hitgroup);
	pipelineDesc.AddShaderResourceView(p_IndexBuffer->GetBuffer(), 2u, 0u,
Skateboard::RaytracingShaderVisibility_Local_Hitgroup);

pipelineDesc.AddConstantBufferView(Skateboard::MemoryManager::GetUploadBuffer(m_
PassBuffer), 0u, 0u, Skateboard::RaytracingShaderVisibility_Local_RayGen);
	pipelineDesc.AddDescriptorTable(p_TextureTable.get(),
Skateboard::RaytracingShaderVisibility_Local_RayGen);

pipelineDesc.AddSampler(Skateboard::SamplerDesc::InitAsDefaultTextureSampler(0u,
0u, Skateboard::RaytracingShaderVisibility_Local_RayGen));
	p_RaytracingPipeline.reset(Skateboard::RaytracingPipeline::Create(L"Raytracing
Pipeline", pipelineDesc));
}

RaytracingLayer::~RaytracingLayer()
{
}

void RaytracingLayer::OnResize(int newClientWidth, int newClientHeight)
{
}

void RaytracingLayer::OnAttach()
{
}

void RaytracingLayer::OnDetach()
{
}

bool RaytracingLayer::OnHandleInput(float dt)
{
	Skateboard::Input::ControlCamera(m_Camera, dt);
	if (m_Camera.HasMoved())
	{
		RPassBuffer passData = {};
		passData.ViewMatrix = m_Camera.GetViewMatrix();
		passData.ProjectionMatrix = m_Camera.GetProjectionMatrix();
		passData.InvView = glm::inverse(passData.ViewMatrix);
		passData.InvProj = glm::inverse(passData.ProjectionMatrix);
		Skateboard::MemoryManager::UploadData(m_PassBuffer, 0, &passData);
	}
	return true;
}

bool RaytracingLayer::OnUpdate(float dt)
{
	return true;
}

void RaytracingLayer::OnRender()
{
	Skateboard::RenderCommand::DispatchRays(p_RaytracingPipeline.get());

Skateboard::RenderCommand::CopyUAVToBackBuffer(p_RaytracingPipeline->GetOutput
UAV());
}

void RaytracingLayer::OnImGuiRender()
{
}
```

Appendix C - Raytracing.hlsl

```
struct VertexType
{
	float3 position;
	float2 texCoord;
};

struct RadiancePayload
{
	float2 uv;
	uint instanceID;
	bool hit;
};

struct PassBuffer
{
	float4x4 ViewMatrix;
	float4x4 ProjectionMatrix;
	float4x4 ViewMatrixInverse;
    float4x4 ProjectionMatrixInverse;
};

RWTexture2D<float4> gOutput : register(u0, space1);                                    //
Raytracing output texture, supplied by the engine
RaytracingAccelerationStructure gSceneBVH : register(t0, space0);
ConstantBuffer<PassBuffer> gPassBuffer : register(b0, space0);
StructuredBuffer<VertexType> gVertices : register(t1, space0);
StructuredBuffer<uint> gIndices : register(t2, space0);
//StructuredBuffer<uint> gVertexOffsets : register(t3, space0);
//StructuredBuffer<uint> gIndexOffsets : register(t4, space0);
Texture2D<float4> gTextures[] : register(t0, space2);
SamplerState gSampler : register(s0, space0);

void GenerateCameraRay(out float3 origin, out float3 direction)
{
	// Get the location within the dispatched 2D grid of work items
	// (often maps to pixels, so this could represent a pixel coordinate).
	uint2 launchIndex = DispatchRaysIndex().xy;
	float2 dims = float2(DispatchRaysDimensions().xy);

	// Calculated the floating point pixel coordinate normalised on [0,1] x [0,1]
	float2 pixelCoordNormalised = (((launchIndex.xy + 0.5f) / dims.xy) * 2.f - 1.f);

	// Perspective ray (see hand written notes)
	// Note that the matrices are transposed since the Nvidia camera buffer did not
transpose them on the CPU
	// This could be avoided by pre-multiplying (i.e. mul(viewI, float4(0,0,0,1)); etc.)
	origin = mul(gPassBuffer.ViewMatrixInverse, float4(0, 0, 0, 1)).xyz;
                                                                                  // The
view matrix dictates the origin of the ray. Retrieve the origin by mutiplying the inverse view
with the world origin (0, 0, 0)
	float4 target = mul(gPassBuffer.ProjectionMatrixInverse,
float4(pixelCoordNormalised.x, -pixelCoordNormalised.y, 1, 1));     // Apply the inverse
projection to the pixel coordinate. This creates a target vector from (0,0,0) to pixCoord
	direction = mul(gPassBuffer.ViewMatrixInverse, float4(target.xyz, 0)).xyz;
                                                                                    // The
direction is origin -> pixel coordinate
}

VertexType GetInterpolatedVertex(/*in const uint meshID,*/ in const uint primitiveIndex, in
const float2 bary)
{
	// Calculate gamma
	// Barycentric coordinates: alpha + beta + gamma = 1
	// Thus: gamma = 1 - alpha - beta
	const float alpha = bary.y;
	const float beta = bary.x;
	const float gamma = 1.f - alpha - beta;
	const float3 barycentrics = float3(gamma, beta, alpha);

	// Plan
	// 1) Get the base vertex buffer index and the base index buffer index
	// 2) Retrieve the current primitive indices (3 indices per triangle)
	// 3) Find the 3 vertices of the primitive using the indices and the base vertex buffer
index
	//			-> vertexBufferPos[i] = baseVertexPos + index[i]
	// 4) Apply the barycentric weighting on each vertex and add them all together
	//			-> This will essentially interpolate between them
	//			-> gamma * vertex[0] + beta * vertex[1] + alpha * vertex[2]

	// 1
	// Here we have 0u as our vertex and index buffers only describe the triangle
	// If you are packing your geometry into the same buffers, then you would
	// need to consider their offsets instead
	const uint baseVertexLocation = /*gVertexOffsets[meshID]*/ 0u;
	const uint baseIndexLocation = /*gIndexOffsets[meshID]*/ 0u;

	// 2
	const uint baseIndexOffset = 3u * primitiveIndex;
	const uint3 primitiveIndices = uint3(
		gIndices[baseIndexLocation + baseIndexOffset + 0u],
		gIndices[baseIndexLocation + baseIndexOffset + 1u],
		gIndices[baseIndexLocation + baseIndexOffset + 2u]
	);

	VertexType vertex = (VertexType)0;
	[unroll] for (uint i = 0; i < 3u; i++)
	{
		// 3
		const uint currentVertexLocation = baseVertexLocation + primitiveIndices[i];
		VertexType temp = gVertices[currentVertexLocation];

		// 4
		vertex.position += barycentrics[i] * temp.position;
		vertex.texCoord += barycentrics[i] * temp.texCoord;
```

```
			// Other attributes you may supply
			//vertex.normal		+= barycentrics[i] * temp.normal;
			//vertex.tangent		+= barycentrics[i] * temp.tangent;
			//vertex.bitangent	+= barycentrics[i] * temp.bitangent;
		}
		//vertex.normal = normalize(vertex.normal);
		//vertex.tangent = normalize(vertex.tangent);
		//vertex.bitangent = normalize(vertex.bitangent);

		return vertex;
}

[shader("raygeneration")]
void RayGenShader()
{
		// Generate the viewport ray
		float3 rayOrigin, rayDirection;
		GenerateCameraRay(rayOrigin, rayDirection);

		// Trace the ray
		RayDesc ray;
		RadiancePayload rayPayload = (RadiancePayload)0;
		ray.Origin = rayOrigin;
		ray.Direction = rayDirection;
		ray.TMin = .1f;
		ray.TMax = 100000.f;
		TraceRay(
			gSceneBVH,
			RAY_FLAG_NONE,
			0xFF,
			0,
			0,
			0,
			ray,
			rayPayload
		);

		// Calculate a shading colour from the payload information
		float4 output = float4(0.f, 0.f, 1.f, 1.f);
		if(rayPayload.hit)
			output =
gTextures[NonUniformResourceIndex(rayPayload.instanceID)].SampleLevel(gSampler,
rayPayload.uv, 0);

		// Write to the output
		gOutput[DispatchRaysIndex().xy] = output;
}

[shader("closesthit")]
void ClosestHitShader(inout RadiancePayload payload, in
BuiltInTriangleIntersectionAttributes attributes)
{
		// Retrieve the interpolated vertex on our triangle
		VertexType vertex = GetInterpolatedVertex(PrimitiveIndex(),
attributes.barycentrics);

	// Assign the payload
	payload.uv = vertex.texCoord;
	payload.instanceID = InstanceID();
	payload.hit = true;
}

[shader("miss")]
void MissShader(inout RadiancePayload payload)
{
	payload.hit = false;
}
```