Laboratório VISGRAF
Instituto de Matemática Pura e Aplicada

Proceduray:
A light-weight engine for procedural primitive ray tracing

Vinicius da Silva, Tiago Novello, Helio Lopes and Luiz Velho

Technical Report TR-21-04 Relatório Técnico

January - 2021 - Janeiro

Proceduray – A light-weight engine for procedural primitive ray tracing.

Vinícius da Silva[1], Tiago Novello[2], Hélio Lopes[1], and Luiz Velho[2]

[1]*Pontifical Catholic University of Rio de Janeiro - PUC-Rio, Department of Informatics, Rio de Janeiro, Brazil*
[2]*IMPA - VISGRAF Laboratory, Rio de Janeiro, Brazil*

December 21, 2020

Abstract

A couple of years after its launch, NVidia RTX is established as the standard low-level real-time ray tracing platform. From the start, it came with support for both triangle and procedural primitives. However, the workflow to deal with each primitive type is different in essence. Every triangle geometry uses a built-in intersection shader, resulting in straightforward shader table creation and indexing, analogous to common practice geometry management in game engines. Conversely, procedural geometry applications use intersection shaders that can be as generic or specific as they demand. For example, an intersection shader can be reused by several different primitive types or can be very specialized to deal with a specific geometry. This flexibility imposes strict design choices regarding hit groups, acceleration structures, and shader tables. The result is the difficulty that current game, graphics, and scientific engines have to integrate RTX procedural geometry (or RTX at all) in their workflow. A symptom of this attrition is the current lack of discussion material about RTX host code development. In this paper, we propose Proceduray, a ray-tracing engine with native support for procedural primitives. We also discuss in detail all design choices behind it, aiming to foment the discussion about RTX host code.

1 Introduction

Given the potential of the RTX architecture for real-time graphics applications, important game engines [1, 2], scientific graphics frameworks [3] and 3D creation software [4] hastily incorporated this technology. Currently, all of the cited tools have at least some support for triangle geometry ray tracing using RTX.

1.1 State-of-the-art authorial tools

We use the term *authorial tool* to describe software for creating real-time 3D interactive applications using high level abstractions. The currently available authorial tools supporting RTX are mostly limited to Falcor [3], Unreal [2], and Unity [1]. There are other choices for working with RTX, but they are either lower-level abstraction libraries [5, 6, 7, 8] or non-interactive ray tracers [9].

Each one of the aforementioned tools approaches RTX in a particular way. Falcor [3] has the more straight-forward integration with the platform, providing very informative samples, including code to use ray-generation, closest-hit, any-hit, and miss shaders, and a path tracer (available in version 3.2). It did not support intersection shaders or procedural geometry until version 4.3, debuted in December 2020.

Unity [1] and Unreal [2] have specific development branches [10, 11] with RTX enabled. As Falcor, Unity supports customized RTX shaders [12, 13], but does not work with intersection shaders. Unreal takes a different approach. Even though it has the most sophisticated RTX-based real-time ray tracer of all the alternatives, it can only be used as an effect, which can be turned on or off and be given parameters. We conjecture that Unreal has everything needed to support customized RTX shaders internally, but it currently lacks documentation and examples on the matter [14, 15, 16].

The fact that two years after the RTX launch the majority of the state-of-the-art authorial tools do not support it in its full potential evidence that deeper discussions on the matter are necessary.

1.2 Real-time photo-realist procedural geometry

Although the triangle geometry case is the most common for rendering purposes, RTX also supports procedural geometry. This feature includes control of the intersection behavior, which directly impacts how expressive a ray type can be. An immediate consequence is another level of flexibility available to applications. Additionally, procedural geometry imposes little resource maintenance in comparison with triangles. Device code is directly responsible for defining geometry, instead of just receiving it to process.

The only authorial tool that supports procedural geometry in this context is Falcor and was not available until December 2020, however. This version documents its API for working with procedural geometry, but it does not has examples on the matter yet. Working directly with the RTX runtime is not a very productive option, since it lacks high-level abstractions, usually available in engines. A challenge that emerges

from this discussion is how to deal with procedural geometry's inherent flexibility and be productive at the same time.

1.3 Contributions and proposal

We propose Proceduray, a novel light-weight engine with native support for procedural primitive ray tracing, designed to be a fair compromise between flexibility and productivity. Another objective of this work is to expand the discussion about host code in the RTX architecture. Currently, references about RTX device shader code are abundant [8, 17], but about host code management are very scarce, usually reduced to code samples in low-level abstraction libraries.

We use Proceduray to render images of scenes composed of complex procedural geometries, such as the 3D fractals *Julia sets* and *Mandelbulb*.

The paper is structured as follows. Section 2 presents background concepts, necessary to understand the problems and the discussion. Section 3 describes in detail the problems that Proceduray deals with and the design choices to do so. Sections 4 and 5 discuss the host API and device shaders respectively. Finally, Section 6 concludes and discusses future works.

2 Background

The RTX architecture introduced a new rendering pipeline and several new graphics concepts with it. Since our objective is to write an engine using this technology, it is natural to first understand those ideas and how they relate with well-established ones, previously introduced by the rasterization and compute pipelines.

2.1 Summary

At a high abstraction level, the host application must perform a few tasks before being able to dispatch ray tracing using RTX:

- Define an efficient way to detect ray-geometry intersections;
- Customize behavior based on geometry type. In other words, a specific geometry should run a specific set of shaders;
- Specify the resources needed for the shaders.

Our approach is to organize the background dependency concepts and develop from there until we know in detail how to perform each task. We focus on practical host resource management since device shader is a topic already well covered in other references [8, 17]. Additionally, we use the DirectX 12 Raytracing (DXR) [18] because it is updated and changes tend to arrive there first.

2.2 Graphics Concepts

Our objective in this section is to understand the building blocks needed to make resources available to DXR shaders, their dependencies, and their

relationships. Since we opt to use DXR, the concepts naturally come from DirectX 12 [19] and HLSL [20].

2.2.1 Overview

Every HLSL shader has an associated *Root Signature*, which is one of the most important concepts for shader resource management. The term Signature is not chosen randomly here: a shader Root Signature is analogous to a programming language function signature. As a function signature describes the arguments needed for a function, a Root Signature describes the resources needed for a shader. Figure 1 contains an overview of a Root Signature. We will explain its components in detail in the next sections.

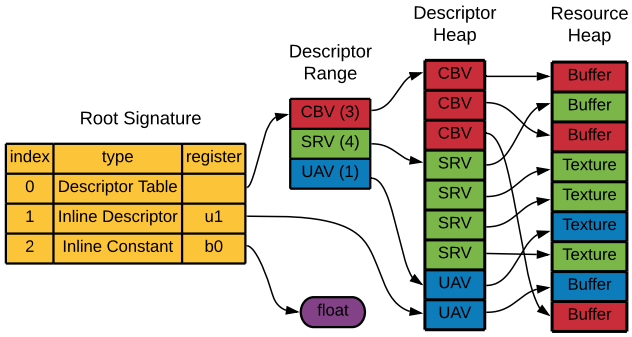

Figure 1: Overview of a Root Signature. Its entries can be Descriptor Tables, Inline Descriptors, or Inline Constants. Read the next Sections for more details.

For didactic purposes, the concepts are organized into two groups: graphics pipeline and ray tracing pipeline. In practice, however, they are mixed in applications.

2.2.2 Graphics pipeline

Resource: It is every non-thread-local memory that is referenced by a shader, such as textures, constant buffers, images, and so on. Differently from thread-local variables, which are stored directly in device registers, *Resource* registers contain indirections for the actual data in a Resource Heap. Figure 1 depicts such an example, where register $u1$ is an indirection to a Buffer. To make a resource available at shader execution time, one needs to use the concepts of *Root Signatures*, *Descriptors*, and *Views*.

Descriptor: *Resources* are not bind directly to the shader pipeline; instead, they are referenced through a *Descriptor*. A *Descriptor* is a small object that contains information about one resource. It resides in a *Descriptor Heap* (see Figure 1).

Descriptor Heap: It is a container for *Descriptors*.

Descriptor Table: It is a set of references to *Descriptors* in *Descriptor Heaps*. It is composed of *Descriptor Ranges*. In Figure 1, entry 0 in the *Root Signature* is a *Descriptor Table*.

Descriptor Range: It is a range of consecutive *Descriptors* in a *Descriptor Heap*. In Figure 1, the *Descriptor Table* entry is composed of 3 *Descriptor Ranges*, one for each *View* type. In the example all ranges are defined in the same *Descriptor Heap* for simplicity, but this is not an imposed restriction.

Views: *Resources* are raw memory and *Views* describe how they can be interpreted. The most common types are:

1. *Constant Buffer Views* (*CBVs*): structured buffers. In practice, they are structs transferred from host to device code.
2. *Shader Resource Views* (*SRVs*): typically wrap textures in a format that shaders can access them.
3. *Unordered Access Views* (*UAVs*): enables the reading and writing to the texture (or other resources) in any order. The other types just support reading or writing, not both.
4. Samplers: encapsulate sampling information for a texture, such as a filter, uv-coordinate, and level-of-detail parameters.

Root Signatures: Before RTX, only *Global Root Signatures* existed: when one was bound, it was visible to all shaders dispatched. Since RTX supports per-geometry customized shaders, *Root Signatures* can also be Local now. *Global Root Signatures* continue to be visible to all shaders, but *Local Root Signatures* can be set up to be used only when a specific geometry is ray-traced.

Figure 1 shows an example of a *Root Signature*, which can be composed by:

1. *Inline Constants*, which are structs inlined directly at the *Root Signature*;
2. *Inline Descriptors*, which are are Descriptors inlined directly at the *Root Signature*;
3. *Descriptor Tables*, which are references to *Descriptors* in *Descriptor Heaps*.

2.2.3 Ray tracing pipeline

Ray: It is the entity that will be used to ray trace. It is composed of a *Payload*, which accompanies it through all the pipeline. This *Payload* is customizable (in practice, can be any plain-old-data (POD) struct) and can be changed in ray tracing shaders.

Acceleration Structure: It is a hierarchical spatial data structure used to accelerate ray-geometry intersections. It is composed of *Bottom-Level Acceleration Structures (BLAS)* and *Top-Level Acceleration Structures (TLAS)*. *BLAS* are the acceleration structures where the geometry lies in and *TLAS* are used for instancing (reusing) *BLAS*. The Acceleration Structure creation is performed by the RTX runtime but can be customized by parameters. *BLAS* creation for triangle and procedural geometry are different. A data structure optimized for triangle culling is constructed for the triangle case, while a simpler data structure enclosing procedural geometry Axis-aligned bounding boxes (AABBs) is constructed for the procedural case. The remaining intersection tests inside the AABBs are responsible of customized *Intersection Shaders*. Figure 2 shows an overview of *Acceleration Structures* and *Shader Tables*, which will be detailed later on.

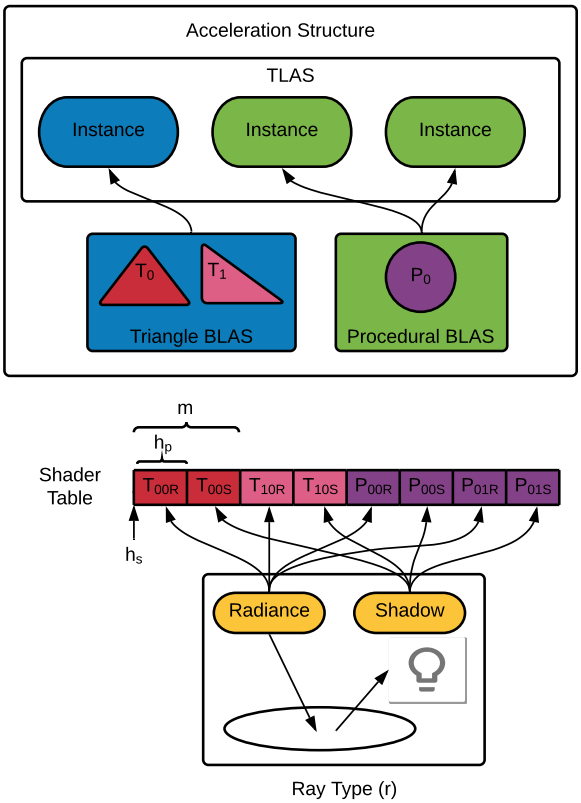

Figure 2: Overview of an *Acceleration Structure*, with associated *Shader Table*. In the example, it is composed of two *Bottom Level Acceleration Structures* (*BLAS*), which are instanced in the *Top Level Acceleration Structure* (*TLAS*). Notice how the Triangle *BLAS* is composed of two geometries. What happens when a node is reached in a traversal depends on the *Shader Table Indexing Rule*. For more details, continue reading this Section.

Ray Tracing Shaders: They are called at different moments of the *Acceleration Structure* traversal by the RTX runtime.

1. *Ray Generation Shader*: is the starting point, where all initial rays are launched. Usually, each ray starts at the camera position and goes through a pixel.
2. *Miss Shader*: is issued when a ray misses all geometry. Receives the ray *Payload* as input.
3. *Any-hit Shader*: is called for all primitive intersections found. Its inputs are the ray *Payload* and a structure coming from the *Intersection Shader*.
4. *Closest-hit Shader*: is called for the primitive intersection closest to the ray origin. It has the same input types as the *Any-hit Shader*.
5. *Intersection Shader*: is called for computing intersections inside a *BLAS* AABB node. Custom intersection shaders are specific to procedural geometry (a built-in optimized *Intersection Shader* is imposed for triangle geometry). The result of the intersection shader indicates if an intersection is found, and the data that must be fed to the *any-hit* and *closest-hit shaders* potentially called by the runtime in consequence.

Hit Group: It is a set of shaders that deal with a specific geometry. Specifically, one or more shaders consisting of: 0 or 1 *Intersection Shader*, 0 or 1 *Any-hit Shader*, 0 or 1 *Closest-hit Shader*. Hit groups are used in conjunction with *Shader Tables* to enable customized behavior for geometries. *Ray Generation* and *Miss Shaders* cannot be part of a *Hit Group* because they aren't involved directly with geometry.

Shader Tables: When traversing the *Acceleration Structure*, the runtime needs to know which *Hit Group* and shader parameters must be used when a *BLAS* node is reached. This information is fed to the pipeline through *Shader Tables*.

A *Shader Table* is composed of *Ray-generation Shader Tables*, *Miss Shader Tables* and *Hit-group Shader Tables*. We are interested in the *hit-group shader* tables, which enable customized behavior for geometries. They are very flexible and their entries must be set according to the *Indexing Rule*. This is the most important (and confusing) concept regarding *Acceleration Structure* behavior and flexibility. It is defined by the following equation:

$$a = h_s + h_p * (r + m * g + i)[18], \tag{1}$$

where:

- a is the address of the *Shader Table* record;
- h_s is the *Hit Group Table* start address;
- h_p is the *Hit Group Table* stride;
- r is the ray type contribution;

- m is a multiplier for the geometry contribution;
- g is the geometry contribution;
- i is the instance contribution.

Figure 2 shows an example. There, h_s is the pointer to the beginning of the table. h_p is the size of each entry. That example has two *Ray* types: Radiance and Shadow. m is set to 2 to reflect this fact. In other words, each geometry has an entry for when it is hit by Radiance *Rays* and another for Shadow *Rays*. With this setup, a *Hit Group* can be used for each case. This is expected since the shadow query is a much simpler operation. g is an ordered index automatically set by the RTX runtime to identify a geometry inside a *BLAS*. In this example, T_0 has index 0 and T_1 has index 1. Finally, i is set by the application when creating the $TLAS$, so the instances can be taken into consideration when indexing. In the example, the Procedural *BLAS* has two instances. If we want each one to be treated differently (which is the case), we set $i = 0$ to the first instance and $i = 1$ to the second instance. To reflect the *Indexing Rule*, each entry in the *Shader Table* of Figure 2 has the form $G_{gi[R|S]}$, where G indicates the Geometry type (T or P), g and i are the parameters in Equation 1 and R or S are possible values for r, indicating the *Ray* type (R for Radiance and S for Shadow). The contents of the entries are defined by the application, but they usually contain data to fetch buffers in the *Global Root Signature*. It is important to note that Equation 1 is very flexible and there is no unique way to use the parameters. A central design choice is how the application deals with the *Indexing Rule*.

One additional property makes Equation 1 a little bit more confusing. The parameters come from elements residing both at the host and at the device code, and each one has a specific way to be set. This is also the main reason why RTX concepts cannot be completely decoupled and should always be thought about regarding their global meaning.

- h_s and h_p must be set by the host code when dispatching the ray tracing device code by calling `DispatchRays()`;
- r and m must be set by HLSL ray tracing shaders when calling `TraceRay()`, the intrinsic responsible for launching a specific ray;
- g is a sequential index automatically generated by the runtime at *BLAS* creation.
- i is set by the host code when creating the *TLAS*.

Ray Tracing Pipeline State Object (RTPSO): *Shader Tables* associate *Acceleration Structures* with *Hit Groups*. However, *Hit Groups* also need an associated *Root Signature* so the *Resources* needed by the shaders are known at execution time. This association is performed by the *RTPSO*, which represents a full set of shaders that could be reached by a dispatch call, with all configuration options and associations resolved. It is composed of *Subobjects*, including:

- *Shader Library (DXIL) Subobject*: contains the compiled shaders;
- *Hit Groups Subobjects*: define the shader entry points;

- *Root Signature Subobjects*: contains *Root Signatures*;
- *Shader Export Association Subjects*: associate *Hit Group Subobjects* with *Local Root Signature Subobjects*.

2.3 Summary revisited

To wrap up, we revisit and update the Summary, given that we now know all the concepts needed to set up ray tracing. The updated tasks before being able to dispatch ray tracing are:

- Creating an *Acceleration Structure* to detect intersections with the geometry.
- Defining a *Shader Table* to indicate which *Hit Group* and shader parameters must be used when ray tracing a given geometry;
- Creating a *RTPSO* to indicate which *Local Root Signature* should be used for a given geometry, which defines all *Resources* needed by the shaders in the *Hit Group*.

3 Proceduray: Design Choices

As could be seen in Section 2, the concepts needed to perform real-time ray tracing using DXR are highly coupled, in special *BLAS*, *Hit Groups*, *Shader Tables*, *Local Root Signatures* and *RTPSOs*. On one hand, we want Proceduray to be flexible to support a variety of procedural geometry applications. On the other hand, too much flexibility can harm the capacity of the engine to provide high-level abstractions that can increase productivity. Proceduray design choices to address those and other difficulties are described next.

3.1 Design choices

3.1.1 New entity types

Since the association between concepts can be intricate, the first design choice is to support several new entity types in scenes, so the user can be flexible in the associations needed by the application. Proceduray supports *Ray*, *Global Root Signature*, *Local Root Signature* and *Hit Group* entities in addition to the usual Geometry entities.

3.1.2 Shader Compatibility Layer

One of the most common problems when writing shaders is to ensure that data transfer between host and device code is robust. Since there is an entire pipeline between the host code and the final image, a lot of time can be lost debugging errors in this area. A usual approach in engines is to fix a compatibility layer for the effects supported by the engine.

To work with procedural shaders, this compatibility layer must be more flexible. Proceduray has a shader compatibility component that centralizes all POD structs that must be seen in host and device code, but

the user defines all of them. The only prerequisite is to classify the structs in predefined categories: *Payloads*, *Root Components*, *Root Arguments* and *Attribute Structs*.

1. *Payloads* are the types used for ray payload in ray tracing shaders.
2. *Root Components* are any type that can be sent in a Root Signature as *Root Constants* or *Root Descriptors*.
3. *Root Arguments* are structs with the actual values that a shader associated with the *Root Signature* will see at execution time.
4. *Attribute Structs* are the types that *intersection shaders* will fill when detecting an intersection so *any-hit* and *closest-hit shaders* receive data to work on.

This design helps the engine to automatically set several parameters needed for the RTX runtime, such as maximum payload, root argument, and attribute struct sizes. It is also the basis for designing Proceduray Type-safe Resources.

3.1.3 Type-safe Resources

Another design choice to ensure security in transfers between host and device code is to ensure type safety for all types declared by the user in the *Shader Compatibility Layer*. Proceduray uses C++ 17 variants to threat all types declared in a single category as a single type engine-wide.

3.1.4 Scene entity queries

Proceduray provides ids for every entity in a scene and fast queries by id using maps. Ids and queries are the core tools to associate entities for *Shader Tables* entries or *RTPSO* Subobjects.

3.1.5 BLAS from Geometry

We choose to create a *BLAS* automatically for each *Geometry* entity, which can be marked as triangle or procedural. Triangle geometry has the usual vertices and indices and Procedural geometry has an AABB only. The instances are defined by the application, at Geometry creation time. The resulting *Acceleration Structure* has a *BLAS* for each *Geometry* and a *TLAS* composed of all instances. The parameter i in the *Indexing Rule* (Eq. 1) is a sequential index depending on the order of Geometry creation.

3.1.6 Shader Table entries decoupled from Shader Tables

To increase flexibility, Proceduray creates *Shader Tables* in two passes. The first is the definition of the table entries, which are created by the user. The second is a building pass that effectively creates the tables for the runtime. Decoupling the entries from the actual table helps to deal with the tight coupling between *Shader Tables* and *RTPSOs*.

3.1.7 Shader table entries as tuples from scene entities

Since we want users to create shader table entries, we facilitate the process by letting the entries be defined as tuples of entity ids and a few additional data. An entry tuple has the form *<Ray id, Hit Group id, Local Root Signature id, Root Arguments>*. The first three parameters are ids to the entities in the scene and the last one is the *Root Arguments* struct. The *Root Arguments* struct contains the actual values that will be received by the shader at execution time. It is important to note that the type safety of the *Root Arguments* struct is automatically ensured by the *Shader Compatibility Layer*.

3.1.8 RTPSO shader export association subojects from Shader Table entries

The *Shader Table* entries are also used to automatically create the association between *Hit Groups* and *Local Root Signatures* in the RTPSO.

4 Proceduray: Host API

Proceduray's design choices reflect in its API, which is discussed in this section. Our approach is to show small simplified code snippets to guide the creation of a simple scene containing a triangle plane mesh and several procedural objects (two CSG Pac-men, a 3D Julia Set, and a Mandelbulb). Figure 3 shows the example.

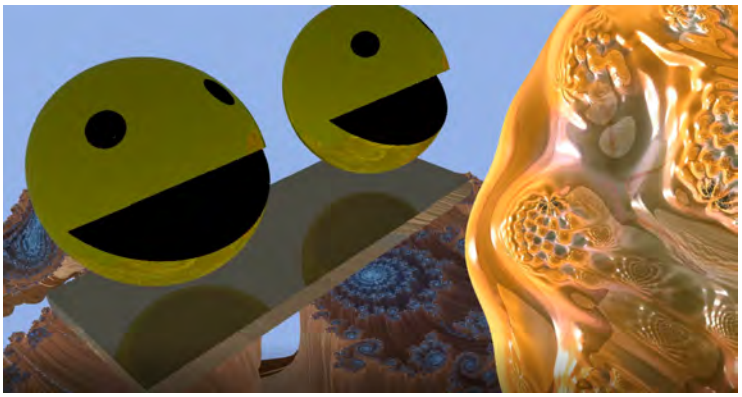

Figure 3: Example scene: a triangle plane mesh and several procedural objects (two Pac-men, a 3D Julia Set and a Mandelbulb).

4.1 Rendering fractals

For completeness, we provide a brief introduction about the fractals rendered in the example scene. Proceduray can render complex procedural geometries as fractals in real-time. Examples of fractals that can be visualized in 3D include Julia sets and Mandelbulbs.

4.1.1 Julia sets

Examples of *Julia sets* arise from the exploration of the convergence of the system given by the iterations of the quadratic function $f(z) = z^2 + c$. Specifically, a (filled-in) Julia set consists of the set of points z_0 in the complexes/quaternions, where the sequence $f^n(z_0)$ has a finite limit. Changing the constant c produces different Julia sets.

Using the complex plane as the domain of the quadratic function f, results in the traditional images of the 2D Julia sets. Norton [21] extended this class of fractals to 4D considering that the quaternions are the domain of f. The resulting 4D Julia set can be seen in the 3D space by taking 3D slices of quaternions.

To visualize a 2D Julia Set, we determine whether a point on the complex plane diverges. Therefore, it suffices to compute the sequence $f^n(z_0)$ and see how quickly its magnitude increases. This test can be applied on points corresponding to pixels in an image, resulting in an illustration of a 2D Julia set.

The above approach is not efficient to render a 3D Julia set. However, as we are interested in looking at the fractal "surface", ray tracing seems to be an appropriate technique. Ray tracing fractals dates back to the work of Hart et al. [22] which uses a distance estimator (described in [21]) to speed up the ray tracing process using *unbounding spheres*. Recently, Quilez [23] presented real-time visualizations of the 3D Julia sets using techniques similar to those defined in [22].

Inspired by the work of Quilez, we used the Proceduray's flexibility to produce visualizations of the 3D Julia set. We generate visualizations based on cutting the 3D Julia set using a plane. See Figure 4.

4.1.2 Mandelbulb

The *Mandelbulb* is a 3D fractal, constructed recently by Daniel White [24] and Paul Nylander [25]. They considered the geometrical properties of the complex numbers (multiplication is related to rotation and addition becomes a movement in a particular direction) to define a kind of "product" of elements in the 3D space. Using this "multiplication" (in 3D) in the polynomial formula $f(z) = z^n + c$ leads us to the Mandelbulb fractal.

We are using unbounding spheres to render the Mandelbulb, so we can control the iterations used to approximate the intersection of the rays with the fractal. See Figure 5.

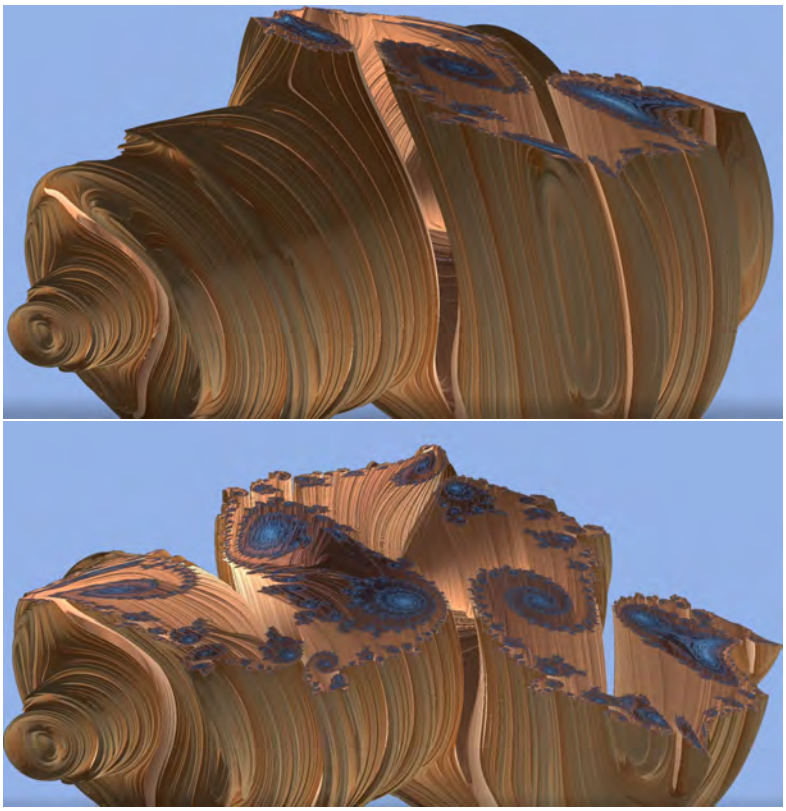

Figure 4: Two Illustrations of the 3D Julia set cut by two different planes. The images restricted to 2D slides are similar to the 2D Julia sets.

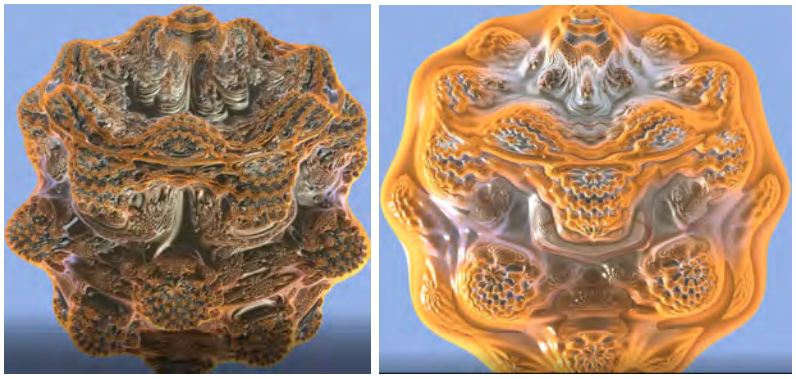

Figure 5: Two frames of an animated Mandelbulb on Proceduray, considering $n = 8$ in the function $f(z) = z^n + c$. The animation is created varying the number of iterations used in the ray marching over time.

4.2 Sample Setup

Overview: Listing 1 shows everything that the sample creates before rendering:

- A *Descriptor Heap* with the desired number of descriptors (3 in this case). Two are used for the triangle mesh vertex and index buffers and the last one is used for the ray tracing output texture.
- The *Rays*;
- The *Hit Groups*;
- The *Geometry*;
- Constant Buffers that allow communication between host and device code;
- An instance buffer that have the per-instance information fetched using the entries in the *Shader Table* and the *Indexing Rule* (Equation 1);
- The output texture;
- The *Acceleration Structure*;
- The *Root Signatures*;
- The *Shader Table* entries;
- The *RTPSO*;
- The *Shader Table*.

Listing 1: Sample main function.

```
CreateSample()
{
        m_descriptorHeap = DescriptorHeap(3);
        CreateRays();
        CreateHitGroups();
        BuildGeometry();
        CreateConstantBuffers();
        CreateInstanceBuffer();
        CreateRaytracingOutputResource();
        CreateAccelerationStructures();
        CreateRootSignatures();
        CreateShaderTablesEntries();

        m_rayTracingState = RayTracingState(
            m_scene,
            m_shaderTable->getCommonEntries(),
            m_descriptorHeap
        );

        m_shaderTable->getBuilded(m_rayTracingState);
}
```

Rays: The user specifies a string id for future queries when creating a *Ray*. This pattern is repeated for almost all entities added to the scene. The API expects the *miss shader* entry point and the *Payload* struct. The example in Listing 2 has a radiance ray type and a shadow ray type.

Listing 2: Ray creation.

```
CreateRays()
{
        m_scene->addRay(
            "Radiance",
            Ray("Miss", Payload(RayPayload()))
        );
        m_scene->addRay(
            "Shadow",
            Ray("Miss_Shadow", Payload(ShadowRayPayload()))
        );
}
```

Hit groups: The API expects a tuple <name, any-hit shader entry point, closest-hit shader entry point, intersection shader entry point> to represent a *Hit Group*. The entry points can be empty if the *Hit Group* does not have the associated shaders. Listing 3 has hit groups for radiance and shadow rays for geometry types. Notice how shadow *Hit Groups* only needs to define the *Intersection Shader* (except for the plane, which must use the built-in one).

Listing 3: Hit group creation.

```

CreateHitGroups()
{
        m_scene->addHitGroup(
            "Triangle", // id in scene
            HitGroup>(
                "HitGroup_Triangle" // internal rtx name,
                "", // any-hit entry point
                "ClosestHit_Triangle", // closest-hit entry point
                "" // intersection entry point
            )
        );

        m_scene->addHitGroup(
            "Triangle_Shadow",
            HitGroup("HitGroup_Triangle_Shadow", "", "", "")
        );

        m_scene->addHitGroup(
            "Pacman",
            HitGroup(
                "HitGroup_Pacman", "",
```

```
                "ClosestHit_Pacman", "Intersection_Pacman"
            )
        );

        m_scene->addHitGroup(
            "Pacman_Shadow",
            HitGroup(
                "HitGroup_Pacman_ShadowRay", "", "",
                "Intersection_Pacman"
            )
    );

        m_scene->addHitGroup(
            "Mandelbulb",
            HitGroup(
                "HitGroup_Mandelbulb", "",
                "ClosestHit_Mandelbulb",
                "Intersection_Mandelbulb"
            )
    );

        m_scene->addHitGroup(
            "Mandelbulb_Shadow",
        HitGroup(
                "HitGroup_Mandelbulb_Shadow", "", "",
                "Intersection_Mandelbulb"
            )
    );

        m_scene->addHitGroup(
            "Julia",
            HitGroup(
                "HitGroup_Julia", "",
                "ClosestHit_Julia",
                "Intersection_Julia"
            )
    );

        m_scene->addHitGroup(
            "Julia_Shadow",
            HitGroup>(
                "HitGroup_Julia_Shadow", "", "",
                "Intersection_Julia"
            )
    );
}
```

Geometry: Triangle meshes expect arrays of vertices and indices, a *DescriptorHeap* and an instance vector as well. The *DescriptorHeap* is used when creating the *Resources* for vertex and index buffers. Procedural geometry just needs an AABB and the instance vector. The instance vector is a sequence of transformation matrices for proper placement of the objects in the scene. Listing 4 shows the code.

Listing 4: Geometry creation.

```
BuildGeometry()
{
    m_scene->addGeometry("GlobalGeometry",
        Geometry(
            vertices, indices, m_descriptorHeap, instances
        )
    );

    AABB juliaAABB{ -3.5f, -3.5f, -3.5f, 3.5f, 3.5f, 3.5f };
    m_scene->addGeometry("Julia",
        Geometry(juliaAABB, juliaInstances)
    );

    AABB pacmanAABB{ -0.5f, -0.5f, -0.5f, 0.5f, 0.5f, 0.5f };
    m_scene->addGeometry("Pacman",
        Geometry(pacmanAABB, pacManInstances)
    );

    AABB mandelbulbAABB{ -1.5f, -1.5f, -1.5f, 1.5f, 1.5f, 1.5f };
    m_scene->addGeometry("Mandelbulb",
        Geometry(mandelbulbAABB, mandelbulbInstances)
    );
}
```

Constant Buffer: It consists of a ConstantBuffer<> template enclosing a POD struct, which is the type exposed to the shader. After association with a *Root Signature*, Proceduray ensures that its contents are uploaded to the GPU and that they will be available at shader execution time. Listing 5 shows the buffer in the example scene, containing parameters such as camera and light values.

Listing 5: Constant buffer creation.

```
struct SceneConstantBuffer
{
       XMMATRIX projectionToWorld;
       XMVECTOR cameraPosition;
       XMVECTOR lightPosition;
       XMVECTOR lightAmbientColor;
       XMVECTOR lightDiffuseColor;
       float   reflectance;
       float   elapsedTime;
};

ConstantBuffer<SceneConstantBuffer> m_sceneCB;

CreateConstantBuffers()
{
    m_sceneCB->Create("Scene Constant Buffer");
```

```
}
```

Instance Buffer: in the example scene, the shaders must access the inverse of the instance transformation matrix so they can work at local space instead of world space. This approach simplifies the *Intersection Shaders*. The creation of this buffer is analogous to the scene Constant Buffer. Listing 6 shows the code.

Listing 6: Instance buffer creation.

```

CreateInstanceBuffer()
{
    auto SetTransformForAABB = [&](
        UINT instanceIndex, XMMATRIX& mTransform
    )
    {
        m_instanceBuffer[instanceIndex].localSpaceToBottomLevelAS
            = XMMatrixInverse(mTransform);
    };

    UINT i = 0;
    for (auto geometry : m_scene->getGeometry())
    {
        if (geometry->getType() == Geometry::Procedural)
        {
            for (auto instance : geometry->getInstances())
            {
                SetTransformForAABB(i++, instance);
            }
        }
    }
}
```

Acceleration Structure. The Acceleration structure is created directly from scene geometry, as discussed in Section 3. Thus, the code API for this is very straight-forward, as can be seen in Listing 7.

Listing 7: Acceleration Structure creation.

```

CreateAccelerationStructure()
{
        m_accelerationStruct = AccelerationStructure(m_scene);
}
```

Global Root Signature: It contains everything that should be visible to all shaders or that would be too heavy to put directly in the *Shader Table*. Our example (Listing 8) defines a *Global Root Signature* with the

output texture, the acceleration structure, the scene constant buffer, the plane, and the instance buffer. Each entry describes which register will be bound to which resource. At shader execution time they will be available using that registers. Notice how the API needs a type specification for each inline entry. This type is ensured by the *Shader Compatibility Layer* for all operations involving those components.

Listing 8: Global Root Signature.

```

CreateGlobalRootSignature()
{
        globalSignature = RootSignature(
            m_descriptorHeap,
            false // is Local Root Signature?
        );

        // === Descriptor ranges. ====

        outputRange = globalSignature->createRange(
            m_raytracingOutputHandles.gpu, // resource handle
            RootSignature::UAV, // entry type
            0, // register index
            1 // range size
        );

        plane = m_scene->getGeometryMap().at("GlobalGeometry");
        vertexRange = globalSignature->createRange(
            plane->getIndexBuffer().gpuDescriptorHandle,
            RootSignature::SRV,
            1,
            2
        );

        // === Entries. ===

        m_raytracingOutputHandles.baseHandleIndex =
            globalSignature->addDescriptorTable(
                vector<RootSignature::DescriptorRange>{outputRange}
        ); // Descriptor table for output texture.

        globalSignature->addEntry(
            RootComponent(DontApply()),
            RootSignature::SRV,
            m_accelerationStruct->getBuilded(),
            0
        ); // Inline Descriptor for acceleration structure.

        globalSignature->addEntry(
            RootComponent(SceneConstantBuffer()),
            RootSignature::CBV,
            m_sceneCB,
            0
        ); // Inline Descriptor for scene constant buffer.

```

```
        globalSignature->addEntry(
            RootComponent(InstanceBuffer()),
            RootSignature::SRV,
            m_instanceBuffer,
            3
        ); // Inline Descriptor for primitive attrib buffer.

        globalSignature->addDescriptorTable(
            vector<RootSignature::DescriptorRange>{vertexRange}
        ); // Descriptor table for plane geometry.

        m_scene->addGlobalSignature(globalSignature);
}
```

Local Root Signatures: The example scene has two *Local Root Signatures*, one for triangles and another for procedural geometry (Listing 9). Type safety is also applied for all types set in the process. The conjunction of the *Shader Table* and the *RTPSO* ensures that the correct *Root Signature* is used for each instance. Notice the use of method `setRootArgumentsType()`, which sets the struct type of the data in the *Root Signature*. The actual values for these types are provided by the *Shader Table* entries, fetched at traversal time.

Listing 9: Local Root Signatures.

```

CreateLocalRootSignatures()
{
    // Triangle geometry local root signature.
    triangleSignature = RootSignature(
    m_descriptorHeap,
        true // is Local Root Signature?
    );

    triangleSignature->addConstant(
        RootComponent(PrimitiveConstantBuffer()),
        1 // register
    );

    // Root Arguments type.
    triangleSignature->setRootArgumentsType(
        RootArguments(TriangleRootArguments())
    );

    m_scene->addLocalSignature("Triangle", triangleSignature);

    // Procedural geometry local root signature.
    proceduralSignature = RootSignature(
        m_descriptorHeap,
        true
    );

    proceduralSignature->addConstant(
```

```
        RootComponent(PrimitiveConstantBuffer()), 1
    );

    proceduralSignature->addConstant(
        RootComponent(PrimitiveInstanceConstantBuffer()), 2
    );

    // Root Arguments.
    proceduralSignature->setRootArgumentsType(
        RootArguments(ProceduralRootArguments())
    );

    m_scene->addLocalSignature("Procedural", proceduralSignature);
}
```

Shader Table entries: They are used to build both the *Shader Table* and the *RTPSO*. The *Shader Table* is responsible for associating *Hit groups* and *Local Root Signature Arguments* with the *Acceleration Structure*, so customized data can be fetch in the traversal. This example creates an entry for each instance, containing values to index the instance buffer in the *Global Root Signature* (Listing 10). Remember that a *Shader Table* entry has the format *<Ray id, Hit Group id, Local Root Signature id, Root Arguments>*.

Listing 10: Shader table entries.

```

void CreateShaderTablesEntries()
{
    m_shaderTable = ShaderTable(m_scene);

    //===== Ray gen. =====//
    m_shaderTable->addRayGen("Raygen"); //id in scene.

    //===== Miss. =====//
    m_shaderTable->addMiss("Radiance");
    m_shaderTable->addMiss("Shadow");

    //===== Triangle Hit Groups. =====//
    TriangleRootArguments rootArgs{ m_planeMaterialCB };
    m_shaderTable->addCommonEntry(
        ShaderTableEntry{
            "Radiance", // ray id.
            "Triangle", // hit group id.
            "Triangle", // local root signature id.
            rootArgs } // Root Argument values.
    );

    m_shaderTable->addCommonEntry(
        ShaderTableEntry{
            "Shadow",
            "Triangle_Shadow",
            "Triangle",
```

```
28          rootArgs }
29      );
30
31      //===== Procedural hit groups. =====//
32      UINT primitiveIndex = 0;
33      UINT instanceIndex = 0;
34
35      UINT i = 0;
36      for (auto geometry : m_scene->getGeometry())
37      {
38          if (geometry->getType() == Geometry::Procedural)
39          {
40              for (auto instance : geometry->getInstances())
41              {
42                  ProceduralRootArguments rootArgs;
43                  rootArgs.materialCb = m_aabbMaterialCB[primitiveIndex];
44                  rootArgs.aabbCB.primitiveType = primitiveIndex;
45                  rootArgs.aabbCB.instanceIndex = instanceIndex;
46
47                  string radianceHitGroup;
48                  string shadowHitGroup;
49
50                  switch (primitiveIndex)
51                  {
52                      case SignedDistancePrimitive::Mandelbulb:
53                      {
54                          radianceHitGroup = "Mandelbulb";
55                          shadowHitGroup = "Mandelbulb_Shadow";
56                          break;
57                      }
58                      case SignedDistancePrimitive::IntersectedRoundCube:
59                      {
60                          radianceHitGroup = "Pacman";
61                          shadowHitGroup = "Pacman_Shadow";
62                          break;
63                      }
64                      case SignedDistancePrimitive::JuliaSets:
65                      {
66                          radianceHitGroup = "Julia";
67                          shadowHitGroup = "Julia_Shadow";
68                          break;
69                      }
70                  }
71
72                  m_shaderTable->addCommonEntry(
73                      ShaderTableEntry{
74                          "Radiance",
75                          radianceHitGroup,
76                          "Procedural",
77                          rootArgs }
78                  );
79                  m_shaderTable->addCommonEntry(
80                      ShaderTableEntry{
81                          "Shadow",
82                          shadowHitGroup,
83                          "Procedural",
84                          rootArgs }
```

```
                );

                ++instanceIndex;
                }
                ++primitiveIndex;
        }
    }
}
```

4.3 Update and Rendering

The setup ensures that every buffer is automatically transferred before rendering, thus the application just needs to change the struct values on an update loop and the changes will be seen at shader execution time. The rendering device code is dispatched by the RTPSO (Listing 11).

Listing 11: Issuing ray tracing.

```

OnRender()
{
    builded_table = m_shaderTable->getBuilded(m_rayTracingState)
    m_rayTracingState->doRayTracing(
        builded_table, // Builded shader table.
        m_width, // Window width.
        m_height // Window height.
    );

    CopyRaytracingOutputToBackbuffer();
}
```

5 Proceduray: Shaders

The goal of the host code is to ensure the availability of all resources at shader runtime. In this section, we show several shaders for the geometries in the example scene and how they relate with the host setup.

Variable declaration (Listing 12): It reflects the *Global and Local Root Signatures* setup in Listings 8 and 9. Note how the register identifiers used (t0 for the first SRV register, u0 for the first UAV register, b0 for the first CBV register, and so forth) map to the host code. Also, notice the *Local Root Signature* variables: `l_materialCB` is used for both triangle and procedural *Local Root Signatures*, while `l_aabbCB` is used to index `g_instanceBuffer`, for the procedural case only. The value for both come from the *Shader Table* entries.

Listing 12: Shader variable declarations.

```
1
2  // Global Root Signature: Acceleration, Textures, and Buffers
3  RaytracingAccelerationStructure g_scene : register(t0, space0);
4  RWTexture2D<float4> g_renderTarget : register(u0);
5  ConstantBuffer<SceneConstantBuffer> g_sceneCB : register(b0);
6
7  // Global Root Signature: Plane buffers
8  ByteAddressBuffer g_indices : register(t1, space0);
9  StructuredBuffer<Vertex> g_vertices : register(t2, space0);
10
11 // Global Root Signature: Procedural resources
12 StructuredBuffer<InstanceBuffer>
13     g_instanceBuffer : register(t3, space0);
14
15 // Local Root Signature
16 ConstantBuffer<PrimitiveConstantBuffer> l_materialCB : register(b1);
17 ConstantBuffer<PrimitiveInstanceConstantBuffer> l_aabbCB: register(b2);
```

Ray-generation shader (Listing 13). It is responsible for creating the initial rays, using the pixel coordinates and camera properties to calculate their origin and direction. `TraceRadianceRay()` is a function used to issue *Radiance Rays* for all geometries (Listing 14).

Listing 13: Ray generation shader.

```
1
2  [shader("raygeneration")]
3  void Raygen()
4  {
5      // Create a ray from camera to pixel.
6      Ray ray = GenerateCameraRay(
7          DispatchRaysIndex().xy,
8          g_sceneCB.cameraPosition.xyz,
9          g_sceneCB.projectionToWorld
10     );
11
12     // Cast a ray into the scene and retrieve a shaded color.
13     UINT currentRecursionDepth = 0;
14     float4 color = TraceRadianceRay(ray, currentRecursionDepth);
15
16     // Write the raytraced color to the output texture.
17     g_renderTarget[DispatchRaysIndex().xy] = color;
18 }
```

Listing 14: TraceRadianceRay() function.

```
1
2  float4 TraceRadianceRay(
3      in Ray ray,
4      in UINT currentRayRecursionDepth,
5      float4 color = float4(0.f, 0.f, 0.f, 0.f))
```

```
{
    if (currentRayRecursionDepth >= MAX_RAY_RECURSION_DEPTH)
    {
        return float4(0, 0, 0, 0);
    }

    // Ray definition.
    RayDesc rayDesc;
    rayDesc.Origin = ray.origin;
    rayDesc.Direction = ray.direction;

    // Parametric ray range.
    rayDesc.TMin = 0;
    rayDesc.TMax = 10000;

    RayPayload rayPayload = {
        color,
        currentRayRecursionDepth + 1
    };

    TraceRay(g_scene,
        RAY_FLAG_CULL_BACK_FACING_TRIANGLES,
        ~0, // all instances
        0, // $r$ in Equation 1
        2, // $m$ in Equation 1
        0, // Radiance ray miss shader index.
        rayDesc,
        rayPayload);

    return rayPayload.color;
}
```

Intersection shader. All procedural geometry intersection tests in the example scene perform 5 tasks:

- Transform the ray to AABB local space;
- Use the signed distance function of the object to detect an intersection and compute the parameter t where it occurs;
- Compute the normal at the intersection and optionally a material color;
- Transform the ray back to world space;
- Report the hit.

Listing 15 shows the code for the Julia Set. To change the shader for Mandelbulb and Pac-man, the function `JuliaDistance()` must be changed to others modeling the desired distances. The details of the distance functions are beyond the scope of this work, but we refer to Inigo Quilez's articles [26, 27, 23] about the subject. We used the distance functions defined there to create the shaders and images in this work.

Listing 15: Intersection shader.

```
1
2  [shader("intersection")]
3  void Intersection_Julia()
4  {
5      Ray localRay = GetRayInAABBPrimitiveLocalSpace();
6
7      float2 thit;
8      ProceduralPrimitiveAttributes attr = {
9          {0.f, 0.f, 0.f}, // normal
10         {0.f, 0.f, 0.f, 1.f} // optional color
11     };
12
13     float3 pos;
14     bool primitiveTest = JuliaDistance(
15         localRay.origin, // input ray origin
16         localRay.direction, // input ray direction
17         attr.normal, // output normal
18         thit, // output ray parameter t
19         g_sceneCB.elapsedTime // input animation parameter
20     );
21
22     // RayTCurrent() is updated by the RTX runtime as
23     // intersections are reported, so the intersection
24     // in other objects are taken into consideration.
25     if (primitiveTest && thit.x < RayTCurrent())
26     {
27         InstanceBuffer aabbAttribute =
28             g_instanceBuffer[l_aabbCB.instanceIndex];
29
30         attr.normal = normalize(
31             mul(attr.normal, (float3x3) WorldToObject3x4())
32         );
33
34         // Using the color parameter to send intersection
35         // min and max t.
36         attr.color = float4(thit, 0.f, 0.f);
37
38         ReportHit(thit.x, /*hitKind*/ 0, attr);
39     }
40 }
41
42 Ray GetRayInAABBPrimitiveLocalSpace()
43 {
44     InstanceBuffer attr = g_instanceBuffer[l_aabbCB.instanceIndex];
45
46     Ray ray;
47     ray.origin = mul(
48         float4(WorldRayOrigin(), 1),
49         attr.localSpaceToBottomLevelAS
50     ).xyz;
51
52     ray.direction = mul(
53         WorldRayDirection(),
54         (float3x3) attr.localSpaceToBottomLevelAS
55     );
56
57     ray.direction = normalize(ray.direction);
```

```
58      return ray;
59  }
```

Closest-hit Shader: Listing 16 shows the Julia Set case. It uses a traditional approach, defining an object color using the Phong model, combining it with a reflection color, and accumulating it with previous reflections.

Listing 16: Closest-hit shader.

```

[shader("closesthit")]
void ClosestHit_Julia(
    inout RayPayload rayPayload,
    in ProceduralPrimitiveAttributes attr)
{
    float3 hitPosition = HitWorldPosition();

    float3 pos = ObjectRayOrigin() +
        RayTCurrent() * ObjectRayDirection();
    float3 dir = WorldRayDirection();

    float4 albedo = float4(
        3.5 * colorSurface(pos, attr.color.xy), 1.f
    );

    if (rayPayload.recursionDepth == MAX_RAY_RECURSION_DEPTH - 1)
    {
       albedo += 1.65 * step(0.0, abs(pos.y));
    }

    // Reflected component.
    float4 reflectedColor = float4(0, 0, 0, 0);

    float reflecCoef = 0.1;

    // Trace a reflection ray.
    Ray reflectionRay = {
        hitPosition,
        reflect(WorldRayDirection(), attr.normal)
    };

    float4 reflectionColor = TraceRadianceRay(
        reflectionRay,
        rayPayload.recursionDepth
    );

    float3 fresnelR = FresnelReflectanceSchlick(
        WorldRayDirection(),
        attr.normal,
        albedo.xyz
    );

    reflectedColor = reflecCoef
        * float4(fresnelR, 1)
        * reflectionColor;

    float diffuseCoef = 0.6;
    float specularCoef = 0.08;
    float specularPower = 0.2;

    // Calculate final color.
    float4 phongColor = CalculatePhongLighting(
        albedo,
        attr.normal,
```

```
56          false, // shadows disabled.
57          diffuseCoef,
58          specularCoef,
59          specularPower
60      );
61
62      float4 color = phongColor + reflectedColor;
63      color += rayPayload.color;
64
65      rayPayload.color = color;
66  }
```

6 Conclusion

This paper introduced Proceduray, a novel light-weight engine focused on procedural primitive ray tracing. We discussed in detail the problems related with integrating RTX procedural geometry in engines. We also discussed RTX host code in detail, a topic with very scarce references.

This work provides an in-depth example of how to create a real-time scene with non-trivial procedural objects, such as the Mandelbulb and Julia Sets, using Proceduray.

Future works include using Proceduray in applications involving procedural geometry. Particularly, we are interested in finding efficient ways to represent curved rays [28, 29, 30, 31]. Additionally, we are considering supporting Vulkan [7] in future versions.

References

[1] John K Haas. A history of the unity game engine. 2014.

[2] Epic Games. Unreal engine.

[3] Nir Benty, Kai-Hwa Yao, Petrik Clarberg, Lucy Chen, Simon Kallweit, Tim Foley, Matthew Oakes, Conor Lavelle, and Chris Wyman. The Falcor rendering framework, 03 2020. https://github.com/NVIDIAGameWorks/Falcor.

[4] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018.

[5] Morgan McGuire, Peter Shirley, and Chris Wyman. Introduction to real-time ray tracing. In *ACM SIGGRAPH 2019 Courses*, pages 1–155. 2019.

[6] Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al. Optix: a general purpose ray tracing engine. *Acm transactions on graphics (tog)*, 29(4):1–13, 2010.

[7] Graham Sellers and John Kessenich. *Vulkan programming guide: The official guide to learning vulkan*. Addison-Wesley Professional, 2016.

[8] Eric Haines and Tomas Akenine-Möller, editors. *Ray Tracing Gems*. Apress, 2019. http://raytracinggems.com.

[9] Patrick Mours. Accelerating cycles using nvidia rtx, 07 2019. https://code.blender.org/2019/07/accelerating-cycles-using-nvidia-rtx/.

[10] Unity. Unity startup dxr project. https://github.com/Unity-Technologies/SmallOfficeRayTracing, 2020.

[11] NVidia. Nvidia dlss unreal engine integration. https://github.com/NvRTX/UnrealEngine, 2020.

[12] Tianliang Ning. Getting started with directx ray tracing in unity - siggraph 2019.

[13] Tianliang Ning. Getting started with directx ray tracing in unity - siggraph 2019.

[14] Unknown. Custom texture2d in an existing shader, 04 2020. https://forums.unrealengine.com/development-discussion/engine-source-github/1741196-custom-texture2d-in-an-existing-shader.

[15] Unknown. Using a custom intersection shader for ray tracing?, 06 2020. https://answers.unrealengine.com/questions/966354/using-a-custom-intersection-shader-for-ray-tracing.html.

[16] Unknown. Custom dxr shader integration, 07 2020. https://forums.unrealengine.com/development-discussion/engine-source-github/1784862-custom-dxr-shader-integration.

[17] Vinicius Silva and Luiz Velho. Ray tracing virtual reality in falcor : Ray-vr. Technical Report TR-05-2019, VISGRAF Lab - IMPA, 2019.

[18] Microsoft. Directx raytracing (dxr) functional spec, 08 2020. https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html.

[19] Microsoft. Direct3d 12 programming guide, 04 2019. https://docs.microsoft.com/en-us/windows/win32/direct3d12/directx-12-programming-guide.

[20] Microsoft. Programming guide for hlsl, 02 2019. https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-pguide.

[21] Alan Norton. Julia sets in the quaternions. *Computers & graphics*, 13(2):267–278, 1989.

[22] John C Hart, Daniel J Sandin, and Louis H Kauffman. Ray tracing deterministic 3-d fractals. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 289–296, 1989.

[23] Inigo Quilez. 3D Julia sets, 11 2020. https://www.iquilezles.org/www/articles/juliasets3d/juliasets3d.htm.

[24] Daniel White. The unravelling of the real 3d mandelbulb. 2009, 2017. https://www.skytopia.com/project/fractal/mandelbulb.html.

[25] Paul Nylander. Hypercomplex fractals, 2017. http://www.bugman123.com/Hypercomplex/index.html.

[26] Inigo Quilez. Distance to fractals, 1 2004. https://www.iquilezles.org/www/articles/distancefractals/distancefractals.htm.

[27] Inigo Quilez. Mandelbulb, 1 2009. https://www.iquilezles.org/www/articles/mandelbulb/mandelbulb.htm.

[28] Luiz Velho, Vinicius da Silva, and Tiago Novello. Immersive visualization of the classical non-euclidean spaces using real-time ray tracing in vr. In *In Proceedings of the 46th Graphics Interface*, 2020.

[29] Tiago Novello, Vincius Silva, and Luiz Velho. Visualization of nil, sol, and sl2 (r) geometries. *Computers & Graphics*, 2020.

[30] Tiago Novello, Vinicius da Silva, and Luiz Velho. Global illumination of non-euclidean spaces. *Computers & Graphics*, 93:61 – 70, 2020.

[31] Tiago Novello, Vinícius da Silva, and Luiz Velho. Design and visualization of riemannian metrics. *arXiv preprint arXiv:2005.05386*, 2020.