Focus: Using Real-Time Ray Tracing Innovatively for Gameplay in a Puzzle Game

Project in Visual Computing
By Simon Maximilian Fraiss
Registration Number 01425602
to the Faculty of Informatics
at the TU Wien

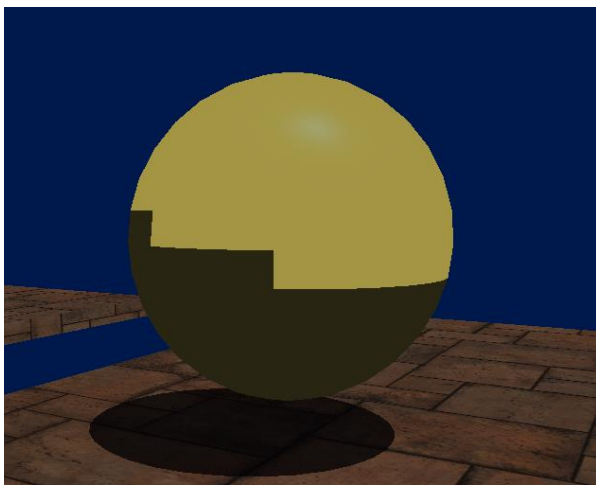
Supervisors:
Univ.Ass. Dipl.-Ing. Johannes Unterguggenberger
Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

---

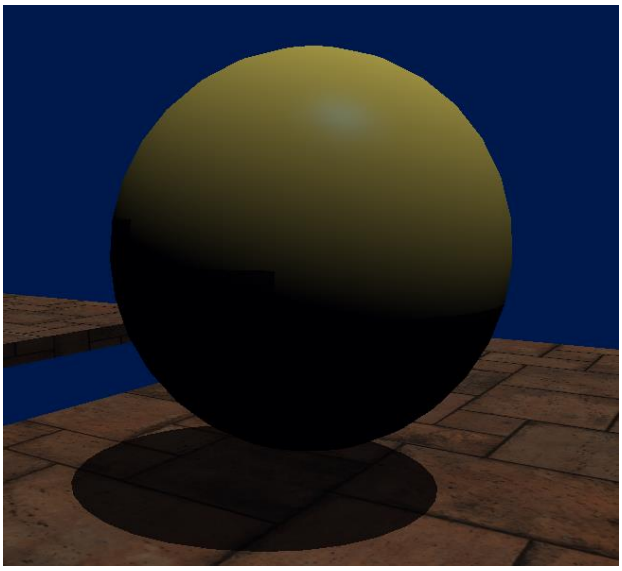
Experienced Pitfalls and Challenges

Unnatural shadows

Calculating shadows using ray tracing is simple enough: Simply shoot a ray from the hit point to the light source and check if there is an intersection. However, on spherical objects the shadows turned out to look very unnatural, as hard edges became visible. The following picture shows the resulting artefacts. For clearer visibility of the problem, the illumination model has been disabled for this screenshot.



Strange as it may look, technically this is correct. The problem is, that the sphere is not a perfect sphere, but is made of polygons. As a polygon is either in the shadow completely or not, these hard edges along the polygon borders become visible. The easiest solution to this was to use the flag `gl_RayFlagsCullBackFacingTrianglesNV` when sending out the shadow ray. This way, back facing polygons are ignored and it is not possible that the front side of the sphere sends a shadow to the backside of the sphere. Of course, the backside of the sphere should still be dark. However, thanks to the classical Blinn-Phong-illumination-model, faces facing away from the light are dark anyway.



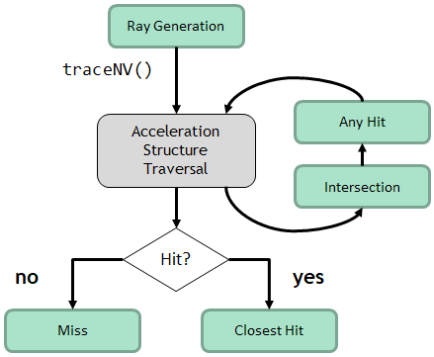
Frames in Flight

One important thing to consider when working with Vulkan, is that we can have several frames being processed at the same time. So when we create an application with moving objects we have to make sure to store the buffers *n* times, where *n* is the number of frames in flight. This way, when we update the data for the current frame, we ensure that we don't change the buffers for one of the previous frames, which might still be processed. Otherwise, jittering of objects may occur. We also have to consider synchronization, as we want to make sure that when we start updating the data for the current frame, the processing of the frame is already done. In pure Vulkan this can be done using Fences. cg_base is supposed to do this automatically, however at the time of the writing of this document, there is still a bug in the framework, which causes synchronization issues and results in occasional jittering of moving platforms. The current status of this issue can be tracked on GitHub [5].

Transparent Objects

One challenge in this project was to render transparent objects. Real-time ray tracing provides so-called any-hit shaders for this task. However, the documentation about them online is very sparse, especially for GLSL, and there are almost no code examples on how to use them. So this needed to be figured out and took some time. Here is how it works:

The acceleration structure traversal on the hardware is working as follows: When an intersection with an object is found, the any-hit shader is called only if the object is not marked as opaque. The any-hit shader receives the payload for the current ray, which will later be passed to the closest-hit shader or the miss shader. The any-hit shader can read from and write into this payload just like the other shaders. Additionally, there are some special functions that can be called in the any-hit shader: The GLSL function `ignoreIntersectionNV` tells the GPU that this intersection should not be considered for the closest hit. The function `terminateRayNV` stops the traversal for the current ray.



Ray tracing shader call structure [4]

Once the traversal is done, the GPU checks if a closest hit has been found (including non-opaque objects, unless `ignoreIntersectionNV` has been called) and then executes either the miss or the closest-hit shader.

Objects can be marked as non-opaque or opaque in several ways. When creating the bottom-level acceleration structure (BLAS) in Vulkan, it is necessary to create a `vk::Geometry` object. If the flag `vk::GeometryFlagBitsNV::eOpaque` is used, then the object is marked as opaque. Otherwise it's non-opaque, so by default the any-hit shader would be called. However, this behavior can be overridden by setting the flags attribute of the corresponding `VkGeometryInstance`, for example by using `vk::GeometryInstanceFlagBitsNV::eForceNoOpaque`. This is how it is done in the game, as this is more flexible and can be changed easily during runtime. Also, this is the only option that cg_base supports at the moment. The opaque-value can also be overwritten by the flags being passed to the function `traceNV` in the shader code. If `gl_RayFlagsOpaqueNV` is used, all objects are assumed to be opaque, so no any-hit shader will be executed, no matter the flags that have been set before. If `gl_RayFlagsNoOpaqueNV` is used, then the any-hit shader will be called for all objects. In our case, we don't want to use either of those flags.

In our scene we have only two transparent objects: The sphere and the character. These objects can be differentiated in the shader by the flags set in their model data. Bearing this in mind, the ray payload in the game is designed to have one color attribute, which will be the final color for the current pixel, as well as two transparent color attributes, which describe the color of each of the two transparent objects – if they hit the ray. Additionally we also store the distances of the transparent object hits (these are initialized in the ray generation shader using a very high value).

If one of the transparent objects is hit, the color at the given point is calculated. This is done in a different way for both objects, as we desire different shading outputs. Both, however, use the dot product of the normal vector and the direction to the eye, in order to have more intense colors in the "inside" of the object than the border. The color and hit distance is simply stored in the corresponding attributes of the payload. However, this only happens if we are on the front side of the object, as otherwise we would potentially override an existing value with a wrong one. We also call `ignoreIntersectionNV`, as we are still interested in solid objects behind the transparent ones.

In the closest-hit shader, we check for both objects if the distance is smaller than the hit distance of the closest hit (if the objects are not on the ray, the distance is surely larger as it has been initialized with a high value). If that is the case, we simply add the color to the color of the closest hit. In the miss shader, these values are always added (if they haven't been set, the default values are 0/0/0, so it doesn't affect the output).

One could also implement alpha blending, by storing alpha values in the payload as well. In that case, the values would have to be blended in the correct order. Additive blending is sufficient in our case.

Checking Sphere Visibility

Another challenge was the checking if the sphere is currently visible in the center of the screen. To do this, the ray payload contains a flag which is only set to one if the sphere is hit. This flag is set in the any-hit shader, as this is the only place where the sphere is processed. As the sphere can also be seen through mirrors, this value has to be passed on when returning from a reflection ray recursion inside the closest hit shader. The ray generation shader then checks if the current pixel is inside the center area of the screen, and if it is, we increment a counter. In OpenGL an atomic counter could be used for this task, but as this does not exist in Vulkan, a simple shader storage buffer object (SSBO) and the

function `atomicAdd` are used. So in the end, the SSBO will contain the number of pixels inside the central area.

On the CPU, the written value is read each frame from the buffer before rendering and used for the sky color and to check the win conditions. However, as we have several frames in flight we can only access the value that is currently written *n* frames later, where *n* is the number of in-flight-frames (usually 3).

Shader Payload Alignment

For testing, we had an NVIDIA RTX 2060, an NVIDIA RTX 2080 and an NVIDIA RTX 2080 Ti available. At some point, all of those provided different results, with some features working and some not. We observed, that by changing the order of the variables in the shader payload the observed behavior of the shader could be changed. It turned out the problem originated from the alignment of the struct member variables of our custom types. As is commonly necessary in other GLSL data structures, such as SSBOs in std140 layout, also the members of the payload data have to be aligned to 16-byte boundaries. An easy way to do this, is to make sure that the size of each member is a multiple of 16 bytes.

So instead of using a payload looking like this:

```
struct RayTracingHit {
	vec4 color;
	vec4 transparentColor[2];
	float transparentDist[2];  //0 = sphere, 1 = character
	uint goal;
	uint recursions;
	bool renderCharacter;
};
```

It should rather look like this:

```
struct RayTracingHit {
	vec4 color;
	vec4 transparentColor[2];
	float transparentDist[4];  //0 = goal, 1 = character
	uvec4 various;             //x = goal, y = recursions, z = renderCharacter
};
```

Taking care of this fixed all payload-related issues.

Gamma Correction

For correct gamma correction two things have to be done. First, sRGB-textures have to be converted back into linear space. Luckily, Vulkan does this automatically, if the format of the texture is specified as an sRGB-format (such as `vk::Format::eR8G8B8A8Srgb`). cg_base's function `cgb::convert_for_gpu_usage` loads color textures automatically as sRGB, if `cgb::settings::gLoadImagesInSrgbFormatByDefault` has been set to `true`. Non-color-textures such as normal maps are never read in with this format, as non-color-data should always be stored in linear space.

Secondly, gamma correction has to be applied during rendering. When creating a Vulkan application with the classical rasterization pipeline, gamma correction is automatically applied, if a surface format of the swap chain is chosen which has its color space as `vk::ColorSpaceKHR::eSrgbNonlinear`. However, this does not work this way when using ray tracing. The reason is that we render into offscreen images and then copy the result into the final framebuffer. When copying values, gamma

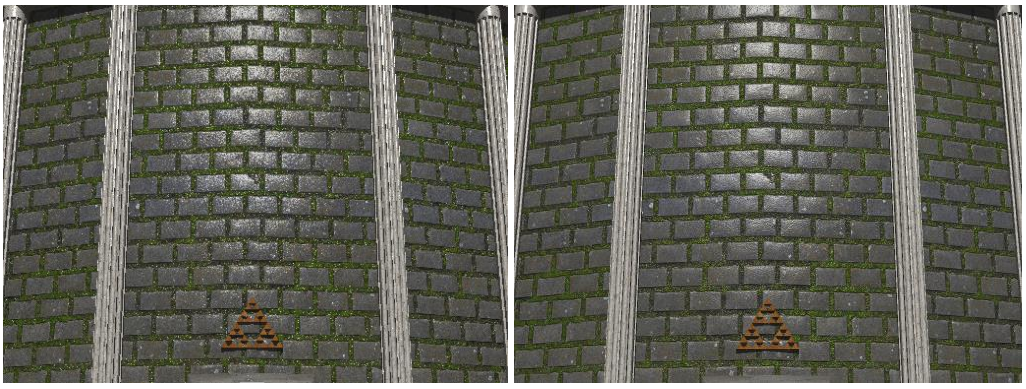
correction is not applied. The solution for this project was to apply gamma correction manually in the ray generation shader. A simple approximation to sRGB-like gamma correction would be to raise the resulting color value to the power of 1/2.2. However, we decided to use the correct sRGB-formula in the shader[1]:

$$\gamma(u) = \begin{cases} 12.92u & u \leq 0.0031308 \\ 1.055u^{1/2.4} - 0.055 & \text{otherwise} \end{cases}$$

As the game was originally designed without working gamma correction (without being aware of it), levels, colors and the illumination model were designed in the wrong color space. This resulted in an unnatural and "greyed-out" look of the game. The main factor for this look was, that in the illumination model the sky color was multiplied into the ambient term in an unnatural way, which made the sky color much too prominent on the objects. Besides changing this, the light intensities of the levels had to be turned down a bit. Fixed colors also had to be changed. For example, the yellow color that appears when changing levels was defined as `(1, 1, 0.5)`. However, with gamma correction this resulted in a color that was too bright and had to be changed to `(1, 1, 0.21)`. Another issue was the sky. Not only changed the colors a little bit, the perlin noise was now too strong. In order to simply achieve the same look as it had before, which looked more natural, the calculations in the miss shader are now simply done the way they were before, but before storing it in the final payload, an inverse gamma correction is applied, so that the later gamma correction will not affect the calculated color anymore.

Antialiasing

One problem with ray tracing is, that it does not support mip mapping, and therefore, heavy aliasing might occur, especially noticeable on objects with strong normal maps. This is also the case in Focus, and it has not been solved yet and is a possibility for future improvements. One simple way to improve the visual quality would be to use supersampling. We could simply render into a bigger render target and then scale the result down to the screen resolution. This was also tested, however it sometimes resulted in low frame rates. The more sophisticated option would be to use Ray Differentials[2] or other techniques[3].



Left: Without supersampling. Right: With supersampling.

References

[1] Wikipedia, sRGB https://en.wikipedia.org/wiki/SRGB

[2] Igehy, Homan. "Tracing ray differentials." https://www.cse.huji.ac.il/~danix/modeling/igehy.pdf

[3] Ray Tracing Gems, Part V, Chapter 20
https://developer.nvidia.com/books/raytracing/raytracing_gems_preview

[4] NVIDIA Developer Blog: Introduction to Real-Time Raytracing with Vulkan
https://devblogs.nvidia.com/vulkan-raytracing/

[5] GitHub: cg_base – Synchronization issues depending on the presentation mode and the number of concurrent frames https://github.com/cg-tuwien/cg_base/issues/16