DV2550 Ray Tracing Project

Christian Markowicz[*]
Blekinge Insitute of Technology

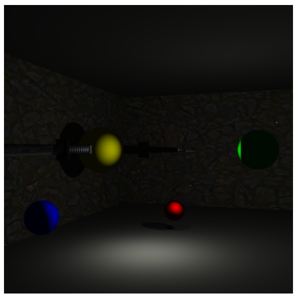

Figure 1: *Screenshot of ray-tracer in action. Seen is a normal mapped stonewall and sphere primitives with one point light and one directional light lighting the scene.*

Abstract

In this paper we have implemented a simple ray-tracer with features such as normal mapping, multiple bounces and super-sampling. We look at the performance analysis and conclude that most of the factors scale linearly.

Keywords: Computer graphics, Real-Time, Interactive computer graphics, Raytrace

Concepts: **•Computing methodologies → Computer graphics;**

1 Introduction

As modern hardware increases in performance, techniques that only were used to perform single frame images are now used in real-time applications. One of those techniques are ray-tracing and is performed by creating rays from the direction of the camera and letting them collide with the scene. In this paper we have implemented a ray-tracer that uses multiple bounces, reflections, shadows, super sampling, normal mapping, triangle and sphere primitives, pointlights and spotlights light sources and support for instancing a vertex object. We look at what variables increase and decrease the performance.

2 Implementation

The overall implementation is based on the Template Win8SDK VS2012 copyright by Stefan Petersson. There are in total 5 shaders used in the implementation. The functionality are as follows; initial ray creation, intersection, coloring, ray creation, super-sampling.

2.1 Interaction

Interaction was implemented in the InputSystem class where in each update call keyboard presses and mouse movement is checked via GetAsyncKeyState and GetCursorPos. The keyboard actions implemented are forward/backward, strafe, up/down, add/remove lights, add/remove bounces, activate/deactivate mouse interaction. The movement and camera interaction is sent to a the Camera class where position, right, up and look vectors are updated. An update call to the camera class that composes the vectors to a view matrix is called once per frame before it is sent to the GPU.

2.2 Creating Rays

The creation of the first rays of the frame is done in the InitRaysCS shader. To create the initial rays the inverse view and projection matrices and pre-calculated screen dimensions factors are stored in a constant buffer. The screen space position is created with the help of the screen dimensions and thread IDs. The screen space directions is created with the help of the screen position and projection matrix. Both the screen space position and direction are then converted into view space with the help of the inverse view matrix. The variables are then placed in a Ray struct that is stored in a structured buffer.

2.3 Intersection

Intersection exist versus sphere and triangles. The spheres are defined with the variables of position, radius and color and are stored in a structured buffer. The vertices are stored in the same fashion with the variables of position, normal, tangent, texture coordinates and material ID. The intersection versus the primitives are implemented in the IntersectionCS shader with collision functions implemented in IntersectionFunctions.fx which is included. We use a float value to store and check the distance to the closest object as we iterate over all the vertices and spheres. As we find a closer object we also store its sphereID or vertexID. The CheckSphereCollision function uses the method described in [Wik a], where the sphere and line equations are combined forming a expression solving the variable of distance. The CheckTriangleCollision uses the function described in [Wik b] and produces both the distance variable but also weight values for each vertex. At the end of the shader function the ID of the closest primitive, hit position, weight values and the data from the Ray struct are stored in a ColorData struct that is stored in a structured buffer.

2.4 Light sources

Light sources exist in two types; pointlights and spotlights, which are stored in structured buffers. Both lights uses variables of position, radius and color. The spotlights has additionally direction and a spot variable that determines the angle to the normal. The

[*]e-mail:christian_markowicz@hotmail.com

coloring is performed in the ColoringCS shader where the illumation functions are described in IlluminationFunctions.fx that is included. The illumination functions are based on the pointlight and spotlight functions described in [Luna 2012] which creates ambient, diffuse and specular lighting. The implementation however uses a slightly different attenuation function which divides the distance to the light source by the radius to remove artefacts as sharp cut-offs. The pointlights are moved on the CPU and updated to the GPU with the directx mapping function.

2.5 Shadows

Shadows are created in the ColoringCS shader and is done by iterating and creating rays to each of the light sources. By using the same functions for intersection in the IntersectionFunctions.fx file we find if there are any intersecting objects. Notable here is that we use the ID of the last primitive collided with to avoid self collision. If no objects collided with the ray the illumination function that were described in the previous section is applied.

2.6 Multiple Bounces

Multiple bounces are achieved by storing more data in the Ray and ColorData structs and using the CreateRaysCS shader as a addition to the previously mentioned shaders. The Ray and ColorData structs also saves the preivous color, reflectionFactor and collided primitive IDs from previous bounce. The primitive IDs are used to avoid self collision in the intersection loop. A for-loop iterating over the number of bounces is used with CreateRaysCS, IntersectionCS and ColoringCS as shaders within. The CreateRaysCS simply transfers the data from the ColorData struct to the Ray struct. There is a trade-off in performance with having this code in the ColoringCS shader as the bounces would be faster but the no-bounce slower.

2.7 Super-sampling

Super-sampling is performed in the SuperSampleCS shader and samples the final color from four ColorData structs and divides the value by four. The shader is dispatched with half the amount of threads as it samples by two in each axis.

2.8 Textures & Materials

Materials are stored in a structured buffer and are pushed to the GPU as they are loaded from an obj file. The material struct consists of ambient color, diffuse color, specular color, specular factor, diffuse texture index and normal texture index. All vertices reference a material which they are given as they are loaded. The textures are stored in Texture2DArray buffers, and can be accessed by index. The tangent of each vertex is computed after load, and are based on [Luna 2012] and [Ras]. In the ColoringCS shader the material is fetched for one of the vertices in the triangle. The normal, and texture coordinates are then computed by the vertex weights from the ColorData struct. The diffuse and normal textures are then sampled with the texture coordinates and index from the material struct. The fetched normal, normal from the vertice and the tangent that is fetched from one of the vertices are then used in a CalculateNormalFromNormalMap function. A matrix is created with the help of the vertex normal and tangent to transform the texture normal to the texture space of the triangle. This normal is then returned and used in the light calculations.

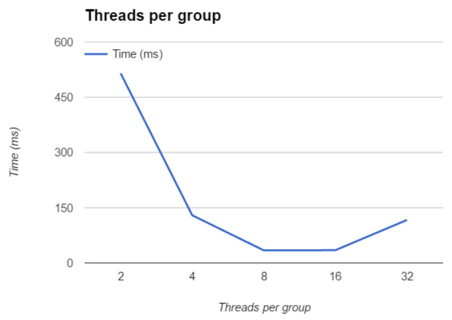

Figure 2: *Frame time in milliseconds by the amount of threads per thread group.*

2.9 Instancing

Instancing of objects is performed by modifying the intersection functions, normal calculation and adding a structured buffer of ObjectInstance struct, that contains a world matrix, start vertex index and stop vertex index. The intersection function and normal calculations now uses a world matrix to transform the vertex positions and normals to world space. The intersection calculations now also iterates over the number of instances which is stored in a constant buffer. For each instance the ObjectInstance struct is fetched and used to iterate over the vertices. Additionally the Ray and ColorData structs also stores InstanceID of last primitive hit to used avoid self intersection.

3 Performance Analysis

The overall performance tests were performed with the scene seen in figure 1 using one directional light and one point light with the resolution of 400x400, no bounces and a vertex count of 990 if the attribute wasn't the variable.

As seen in figure 2, the best performance is achieved by using thread groups of 8 or 16. For the thread groups of 32, there is a possibility that the performance can increase as the application warns about maximum number of registers are achieved.

As screen resolution is a exponential function, we look at the affect of the amount of pixels instead of a screen resolution number. We see in figure 3 that the amount of pixels have a linear dependency with a bit of varying in the end values.

We see that the number of bounces in figure 4 also has a linear affect on the performance.

Seen from figure 5, the amount of light sources also has a seemingly linear affect on the performance, with a bit of varying on 2 and 7 lights.

The amount of vertices seems to at first have a seemingly linear affect on the performance seen in figure 6. However at the last 50 vertices from the diagram we can see that the frame-time increases with more then 100%. This is however possible due to the fact that the last vertices in the buffer contains the outer box seen in figure 1 and thereby increasing the frametime by adding color and light calculations.

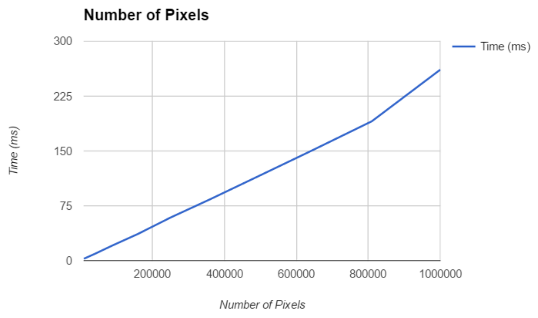

Figure 3: *Frame time in milliseconds by the amount of pixels displayed.*

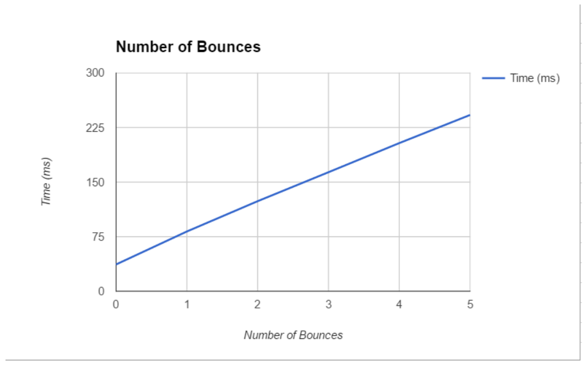

Figure 4: *Frame time in milliseconds by the amount of bounces of rays. Where 0 is no reflections.*

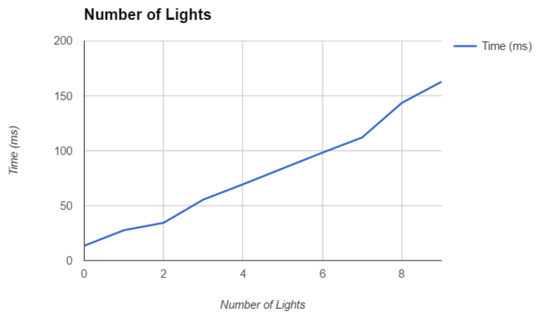

Figure 5: *Frame time in milliseconds by the amount of point lights in the scene.*

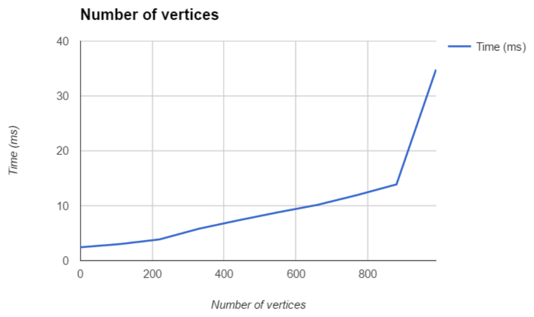

Figure 6: *Frame time in milliseconds by the amount of vertices in the scene.*

4 Conclusions

We conclude that using ray-tracer with interactive frame-rate is possible with the right parameters. We've also seen that pixel amount, light amount, bounce amount and vertex amount all seem to have a linear affect on the performance.

Acknowledgements

I would like to thank Stefan Petersson for the code basis and Konrad Öhman for the given advice during the implementation.

References

LUNA, F. D. 2012. *Introduction to 3D game programming with DirectX 11*. Mercury Learning and Information, Dulles, VA.

Tutorial 20: Bump mapping. http://www.rastertek.com/dx11tut20.html. Accessed: 2017-01-07.

Linesphere intersection. https://en.wikipedia.org/wiki/Line%E2%80%93sphere_intersection. Accessed: 2017-01-07.

Mllertrumbore intersection algorithm. https://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm. Accessed: 2017-01-07.