SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 4030

Ray Tracing Surface Patches

Postupak praćenja zrake na zakrivljenim površinama

Jure Ratković

Zagreb, lipanj 2015.

CONTENTS

LIST OF FIGURES

List of Algorithms

1. Introduction

While drawing one variable functions is strictly limited by the resolution of the screen on which they are drawn, the limits on drawing two variable functions are far more obscure. This paper will cover the standard mesh based approach and propose a new, ray casting approach. These methods will be compared, and their performance and limitations will be commented on. Implementation details will be covered and detailed explanations of implementation decisions and optimizations will be provided. This paper comes with program solution implementing said methods.

2. Function defined surface patches

This paper will be focused on rendering techniques for surface patches defined by two variable functions. All the mathematical analysis will be done in a left-handed coordinate system, the same one OpenGL uses. The input is a user defined function, and an interval on which the function is to be drawn.

$$y = f(x, z), x \in [x_m, x_M], z \in [z_m, z_M]$$

The function must be bounded on the given interval.

$$f(x, z) \in \mathbb{R}, \forall x \in [x_m, x_M], \forall z \in [z_m, z_M]$$

3. Rendering function defined surface patches

We will consider the standard, widely used rasterization method, and the proposed ray casting approach.

3.1. Rasterizing triangle meshes

The standard way of rasterizing two variable functions is to build a heightmap of the function. A planar mesh of $N * M$ vertices is build, and then each vertex' height is set to the value of the function at that point. This ouputs a mesh approximation of the function that can later be rasterized. It is intuitive that as the number of triangles grows, the approximation gets closer to the function, but this approach requires a lot of memory, and is demanding on the vertex processor. Improvements can be made using modern hardware tessellation and geometry shaders, but those techniques are out of scope of this paper.

Algorithm 1 Build function mesh

1: **procedure** BUILDMESH($N, M, f, x_m, x_M, z_m, z_M$)
2: $vertices \leftarrow \emptyset$
3: **for** $i \leftarrow 0$ ... $(N-1)$ **do**
4: **for** $j \leftarrow 0$ ... $(M-1)$ **do**
5: $x \leftarrow i * \frac{(x_M - x_m)}{(N-1)} + x_m$
6: $z \leftarrow j * \frac{(z_M - z_m)}{(M-1)} + z_m$
7: $y \leftarrow f(x, z)$
8: $vertices \leftarrow vertices \cup (x, y, z)$
9: **return** $vertices$

Matlab [1] and Wolfram alpha [2] draw two variable functions this way, as shown in 3.1 and 3.2.

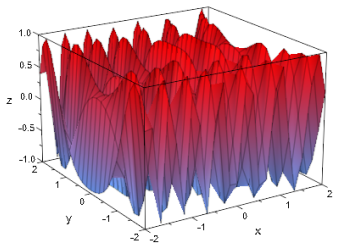

Figure 3.1: Matlab rendering

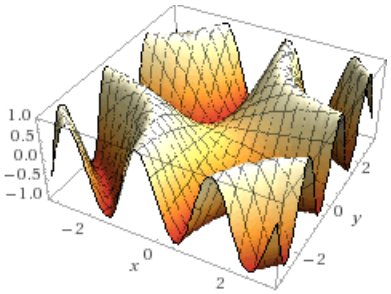

Figure 3.2: Wolfram rendering

3.2. Ray Tracing

Unlike rasterization, ray tracing [3] is a screen space rendering method. Rays whose origin is at eye position are traced trough each screen pixel, and are intersected with objects in the world. At points of intersection, additional rays may be traced to achieve shadows, reflections or refractions, depending on the object's material. This generalization is called recursive ray tracing. Moreover, the algorithm may be further expanded to distributed ray tracing, by tracing multiple rays at each point, allowing for soft shadows, diffuse reflections, fuzzy transparency or multisampling. There are many more ray tracing based algorithms, such as SSAO, photon mapping and path tracing, which allow for high quality image generation, as seen in 3.3. For the purpose of rendering two variable functions, the simplest form of ray tracing, ray casting, will suffice.

Figure 3.3: A ray traced image

3.2.1. Ray casting

Ray casting consists of casting a ray whose origin is at eye position trough each pixel on screen and intersecting that ray with objects on screen.

To compute each ray, we need the camera's local space, consisting of vectors $\mathbf{U}, \mathbf{W}, \mathbf{V}$ (right, direction, up). If we are given camera's horizontal field of view (FoV), aspect ratio (AR), pitch (θ) and yaw (ϕ) the local space is computed as follows:

$$\mathbf{dir} = (\cos(\phi) * \sin(\theta), \sin(\phi), \cos(\phi) * \cos(\theta))$$
$$\mathbf{right} = (-\cos(\phi), 0, \cos(\phi))$$
$$\mathbf{up} = \mathbf{dir} \times \mathbf{right}$$
$$\mathbf{U} = \mathbf{right} * \tan(FoV * \pi) * AR$$
$$\mathbf{W} = \mathbf{dir}$$
$$\mathbf{V} = \mathbf{up} * \tan(FoV * \pi)$$

Let x and y be the indices of the pixel trough which we are casting the ray, and let p_x and p_y be the current screen position mapped to $[-1, 1]$

$$p_x = x/ScreenSize_x * 2 - 1$$
$$p_y = y/ScreenSize_y * 2 - 1$$

Then the normalized direction of the ray is

$$\mathbf{d} = \frac{p_x * \mathbf{U} + p_y * \mathbf{V} + \mathbf{W}}{|p_x * \mathbf{U} + p_y * \mathbf{V} + \mathbf{W}|}$$

Now we can define a ray with origin in eye position $\mathbf{E}$, and a direction $\mathbf{d}$

$$\mathbf{P} = \mathbf{E} + \lambda * \mathbf{d}, \lambda > 0$$

3.2.2. Bisection method

Before explaining the intersection finding algorithm, bisection [4] will be explained as it is a step in the final algorithm. Bisection is a root finding algorithm for one variable

functions. Its input is an interval $[a, b]$, a function $f(x)$, and a tolerance to the distance between a and b, tol. The algorithm outputs the root of f on the given interval if there is one. At each step the midpoint of the interval, c, is computed. Then the sign of the function at midpoint is compared to the sign of the function at the leftmost point of the interval. If the signs do not match, the root is in $[a, c]$, otherwise in $[c, b]$. This process is repeated until we find the root, or the interval becomes too small.

Algorithm 2 Bisection

```
procedure BISECTION(a, b, f)
Require: a < b
    while b − a > tol do
        c ← (a+b)/2
        if f(c) = 0 then
            return c
        if f(a) ∗ f(c) < 0 then
            b ← c
        else
            a ← c
    return c
```

Bisection is relatively slow compared to other root finding algorithms, but at the same time the most stable one. This stability is important as it is crucial that the *closest* intersection between the ray and the function is found at each pixel. In addition, this implementation of bisection assumes that there will always be a root in the input interval, as the root finding algorithm calls bisection only then. A good value for interval length tolerance (tol) is 0.0005, because the starting interval will be small, as we will see in the next chapter.

3.2.3. Ray-function intersection

Now we can define the problem of finding the *closest* intersection between a ray,

$$\mathbf{P} = \mathbf{E} + \lambda * \mathbf{d}, \lambda > 0 \tag{3.1}$$

and a two variable function defined on an interval

$$y = f(x, z), x \in [x_m, x_M], z \in [z_m, z_M] \tag{3.2}$$

Ray (3.1) in parametric form is

$$\begin{cases} x = x_e + \lambda * x_d \\ y = y_e + \lambda * y_d \\ z = z_e + \lambda * z_d \end{cases} \tag{3.3}$$

After inserting (3.3) in (3.2) we get

$$y_e + \lambda * y_d = f(x_e + \lambda * x_d, z_e + \lambda * z_d)$$
$$f(x_e + \lambda * x_d, z_e + \lambda * z_d) - y_e - \lambda * y_d = 0$$

The left side of this equation is a function of λ, $g(\lambda)$.

$$g(\lambda) = f(x_e + \lambda * x_d, z_e + \lambda * z_d) - y_e - \lambda * y_d$$

To find the ray's closest intersection with f, we need to find the minimal (leftmost) root of g, λ_{min}.

Since f is defined on an interval $[x_m, x_M]$, $[z_m, z_M]$ and is bounded on that interval, its axis aligned bounding box can be computed. The first step of the algorithm is intersecting the ray with the functions bounding box. If the ray does not intersect the box the algorithm is terminated. Otherwise, there are two intersections with the box, near (λ_1) and far (λ_2). The algorithm walks from λ_1 to λ_2 using a small step distance, and compares the sign of g between the steps. If the signs are opposite there is a root in between, and it is found using bisection.

Algorithm 3 Ray-function intersection

```
1:  procedure FUNCTIONINTERSECTION(λ1, λ2, λstep)
2:      λa ← λ1
3:      while λa ≤ λ2 do
4:          λb ← λa + λstep
5:          if g(λb) * g(λa) < 0 then
6:              λmin = Bisection(λa, λb, g)
7:              Pi = E + λmin * d
8:              return Pi
9:          λa ← λb
10:     return false
```

4. Implementation details

This paper comes with a program solution that implements said methods, and this chapter will discuss implementation choices. The program was written using C++ [9] and OpenGL [5]. The user inputs a two variable function, it's x and y derivative and an interval on which the function is to be drawn. Derivatives are needed for computing normals, because the function is phong shaded [6], as seen in 4.1. The user can switch between renderers (tracer and rasterizer).

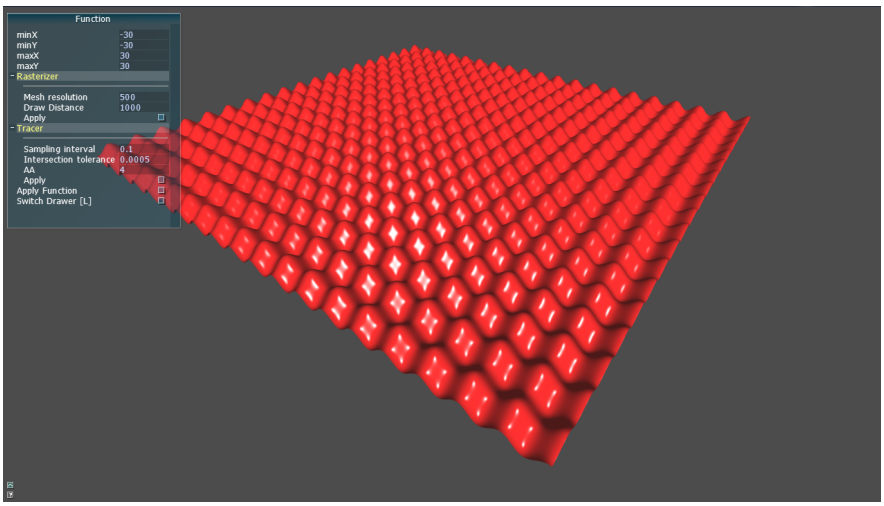

Figure 4.1: Phong shading on function $\sin x^2 + \cos y^2$

4.1. Rasterization

Rasterizer's image quality directly correlates with the number of vertices in the function mesh. Knowing this, we can assume that memory and vertex processor optimizations will be crucial in maximizing the number of vertices.

In fact, maximum number of vertices that the authors implementation supports is $8000 * 8000$, or 64 million vertices, which is much more than two million pixels on a 1080p monitor. That in mind, the vertex shader is minimalistic, with all transformations already baked into the mesh.

Since we want to phong shade the mesh, one might assume that we need two vertex attributes, vertex position and normal. Since those are three component vectors, by using 32 bit floats for vector components we will have 24 bytes per vertex. For a 64 million vertex mesh that equates to approximately 1.5 gigabytes of memory, which is obviously too much. We can alleviate this problem by using 16 bit floats for vertex positions, and fitting the normals into 32 bits, using the *GL_INT_2_10_10_10_REV* format [7]. This equates to 10 bytes per vertex.

But in fact normal as a vertex attribute is not needed. We can paste the function derivatives that the user inputs into the fragment shader before compiling it, and compute the normal from them in the shader:

```
//x derivative
float Fx(in vec2 p)
{
    return #Fx; //paste user input here
}
//y derivative
float Fy(in vec2 p)
{
    return #Fy; //paste user input here
}
//normal
vec3 getNormal(in vec2 p)
{
    return normalize(vec3(-Fx(p), 1.0, -Fy(p)));
}
```

This is indeed faster as there are more vertices in the mesh than pixels on screen, and with this approach the vertex shader does not need to output normals and they do not need to be interpolated between vertices. Also, it improves image quality as we can get the exact normal for any given pixel. Now we have only position with 16 bit floats as a vertex attribute, which is 6 bytes per vertex, or 366 megabytes for 64 million vertices. Any modern GPU should have this much RAM.

Another issue is the fact that the modern graphics API-s do not support such large buffers, so the mesh must be drawn in multiple draw calls. Drawing the mesh as a triangle strip would repeat each row of vertices which would in turn lead to double the memory requirement. Because of this an indexed approach is better suited for the

problem. Mesh can be split into multiple submeshes, each containing the same number of original mesh' rows. Than each submesh can be drawn using the same index buffer. This approach leads to minimal memory redundancy.

In 4.2 and 4.3 we can see the difference in image quality depending on the mesh resolution.

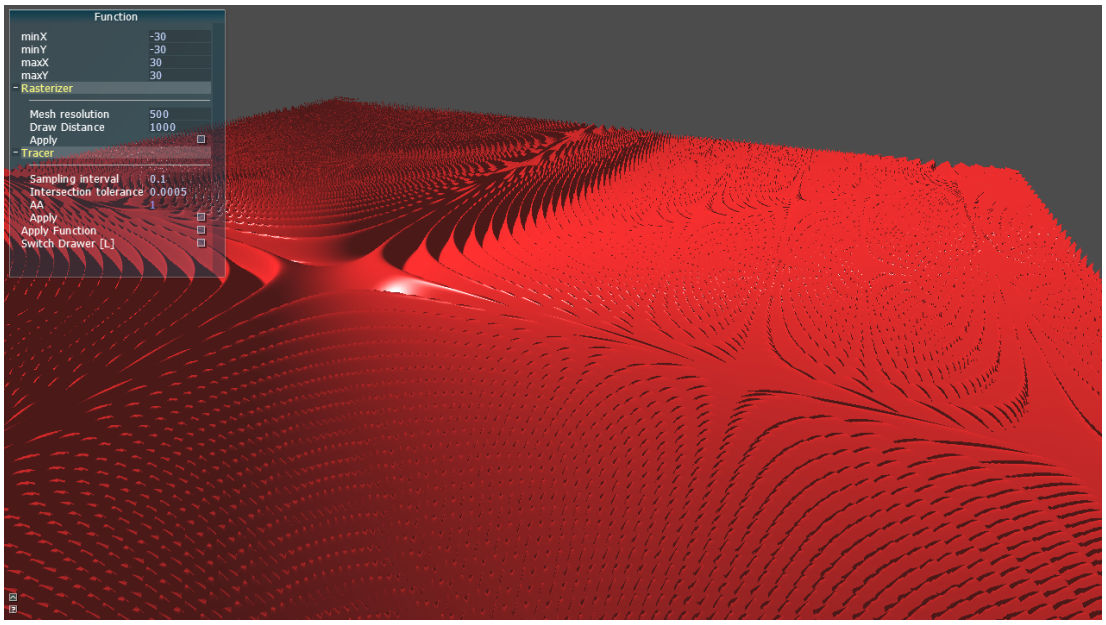

Figure 4.2: $\frac{\sin x*y}{(2+\cos x*y)}$ drawn with $500 * 500$ vertices

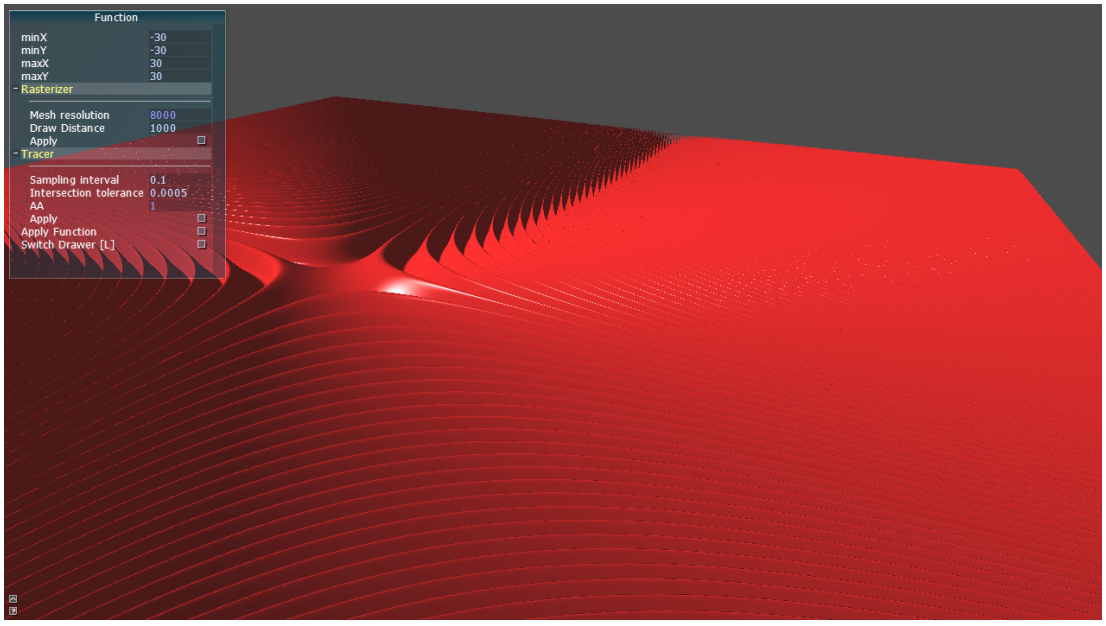

Figure 4.3: $\frac{\sin x*y}{(2+\cos x*y)}$ drawn with $8000 * 8000$ vertices

4.2. Ray tracing

The ray caster is implemented as a GLSL fragment shader. A single quad covering the entire screen is drawn, and the fragment shader determines the color of each pixel by finding the intersection between that pixel's ray and f, and then phong shading it. Similarly to how the normals are implemented in rasterizer, function $g(\lambda)$ is constructed out of the user inputted function and is pasted into the shader.

As these are very intensive tasks, the fragment shader must be as optimized as possible. Control flow divergence must be avoided if possible as threads on the same streaming multiprocessor will have to execute both control paths if divergence occurs. Using control flow statements in bisection (2) and the root finding algorithm (3) can not be avoided, but it can in ray-AABB intersection:

```
//get min and max lambda
bool intersectAABB(in vec3 d, out float Lmin, out float Lmax)
{
    vec3 tmin = (minv - eye) / d;
    vec3 tmax = (maxv - eye) / d;

    vec3 near = min(tmin, tmax);
    vec3 far = max(tmin, tmax);
    Lmin = max(max(near.x, near.y), near.z);
    Lmin = max(0.0, Lmin);
    Lmax = min(min(far.x, far.y), far.z);

    return Lmax >= Lmin;
}
```

Since modern floating point instruction sets can compute min and max without branches, this gives a ray-AABB intersection test with no branches [8].

Main quality settings for the tracer are the sampling interval length (λ_{step}) and anti-aliasing level. Anti-aliasing simply casts multiple rays trough the pixel and averages the results. Effects of anti-aliasing can be seen in 4.4 and 4.5.

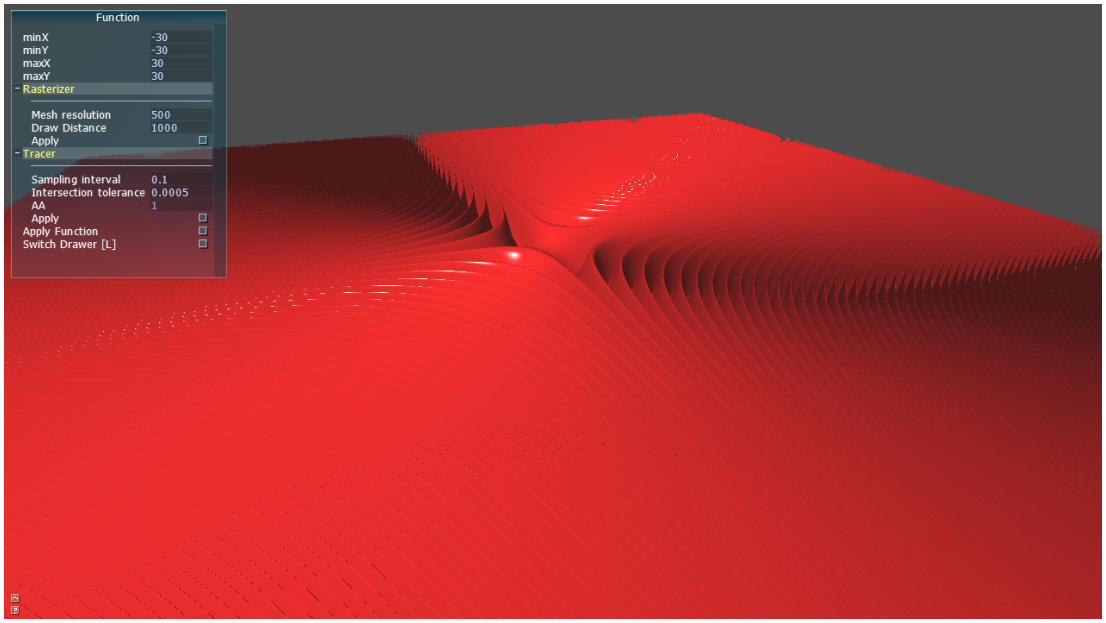

Figure 4.4: $1 * AA$

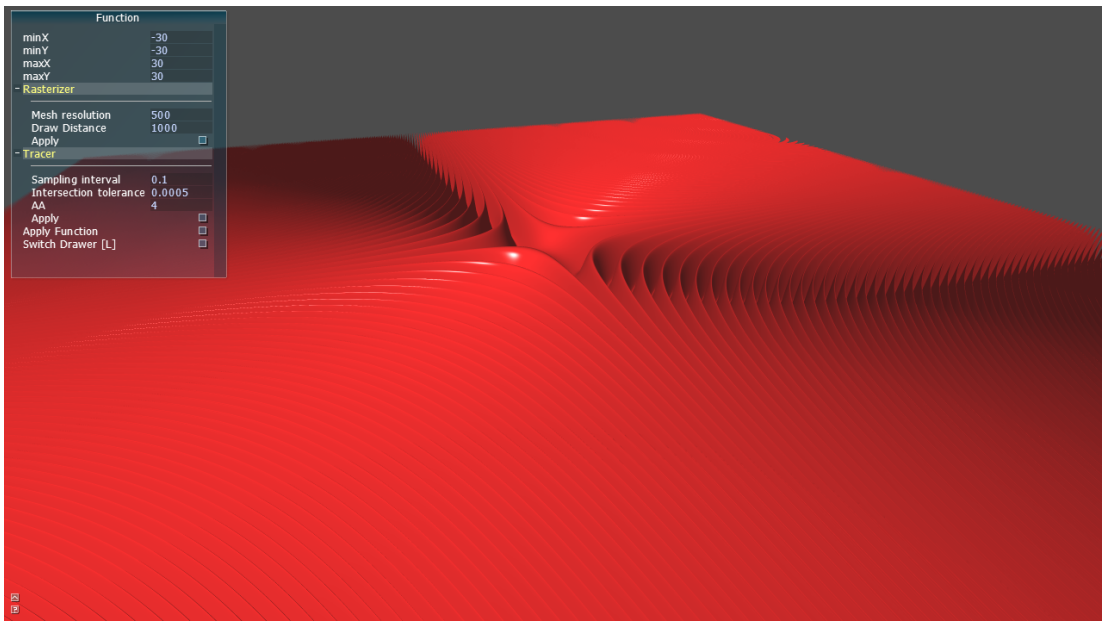

Figure 4.5: $4 * AA$

Decreasing the sampling interval length makes the root search more thorough and gives better results on rapidly changing parts of the function, as shown in 4.6 and 4.7.

Figure 4.6: $\sin x * y, \lambda_{step} = 0.1$

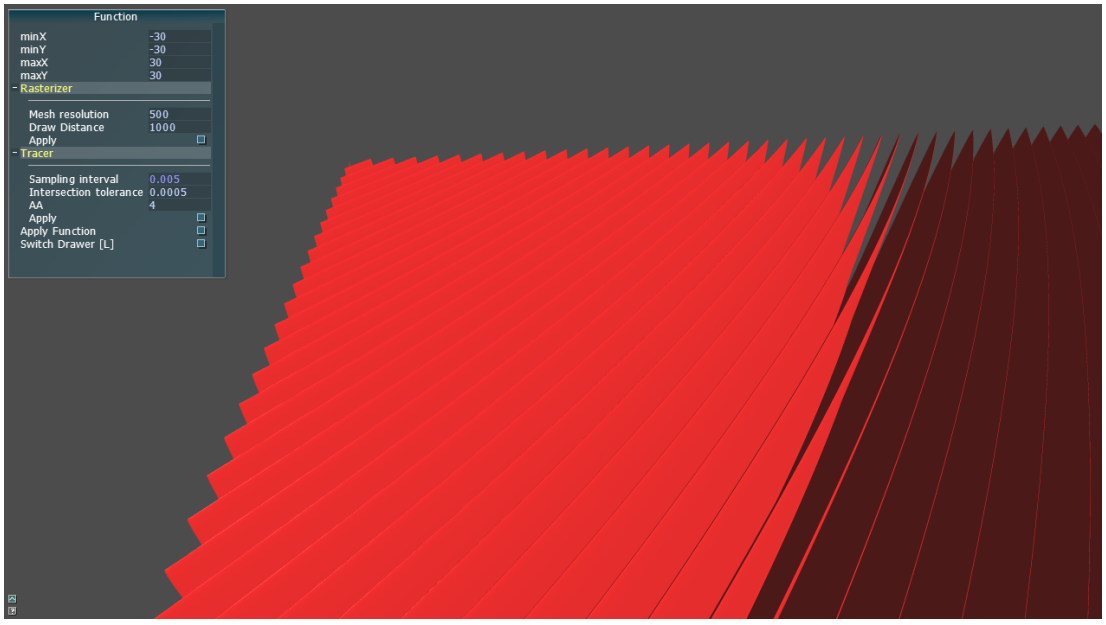

Figure 4.7: $\sin x * y, \lambda_{step} = 0.005$

4.3. Result comparison

The ray tracing approach has better image quality than rasterization approach. Steep function minima and maxima do not have artifacts produced by lack of geometry, as seen in 4.9 and 4.8.

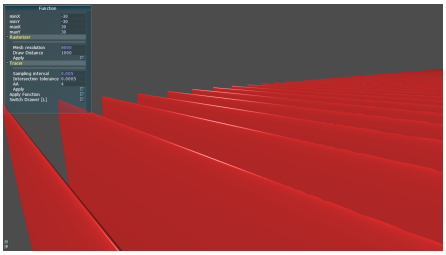

Figure 4.8: Tracer

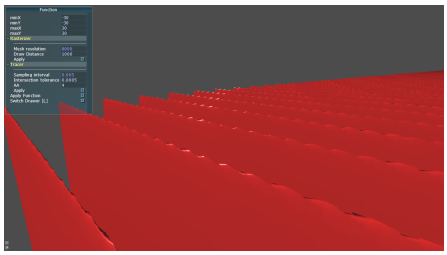

Figure 4.9: Rasterizer

Furthermore, tracer's image quality does not correlate to the size of the domain on which the function is drawn. Images 4.10 and 4.11 show how rasterizer's geometry density is not sufficient for large domains.

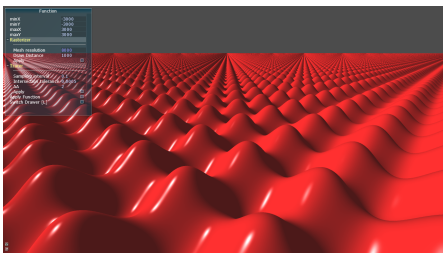

Figure 4.10: Tracer

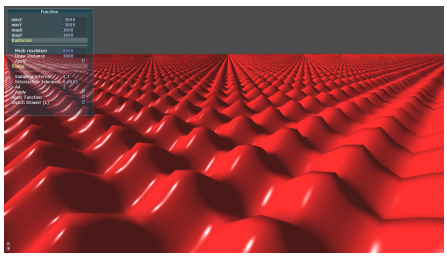

Figure 4.11: Rasterizer

In both examples above, the mesh resolution for rasterizer is $8000 * 8000$ vertices, and its net memory consumption is 412 MB, while tracer uses constant memory. Unlike the rasterizer, tracer does not entail any precomputation time, because it does not use any geometry. Also, if anti aliasing is not used, tracer shows better performance on all test configurations. In the histogram below (4.12), we can see a performance comparison between the tracer and rasterizer on different hardware. The test settings were:

- Screen resolution: $1280 * 720$
- Vsync: on
- Anti aliasing: 1x
- Mesh resolution: $8000 * 8000$
- Bisection interval tolerance: 0.0005
- λ_{step}: 0.01

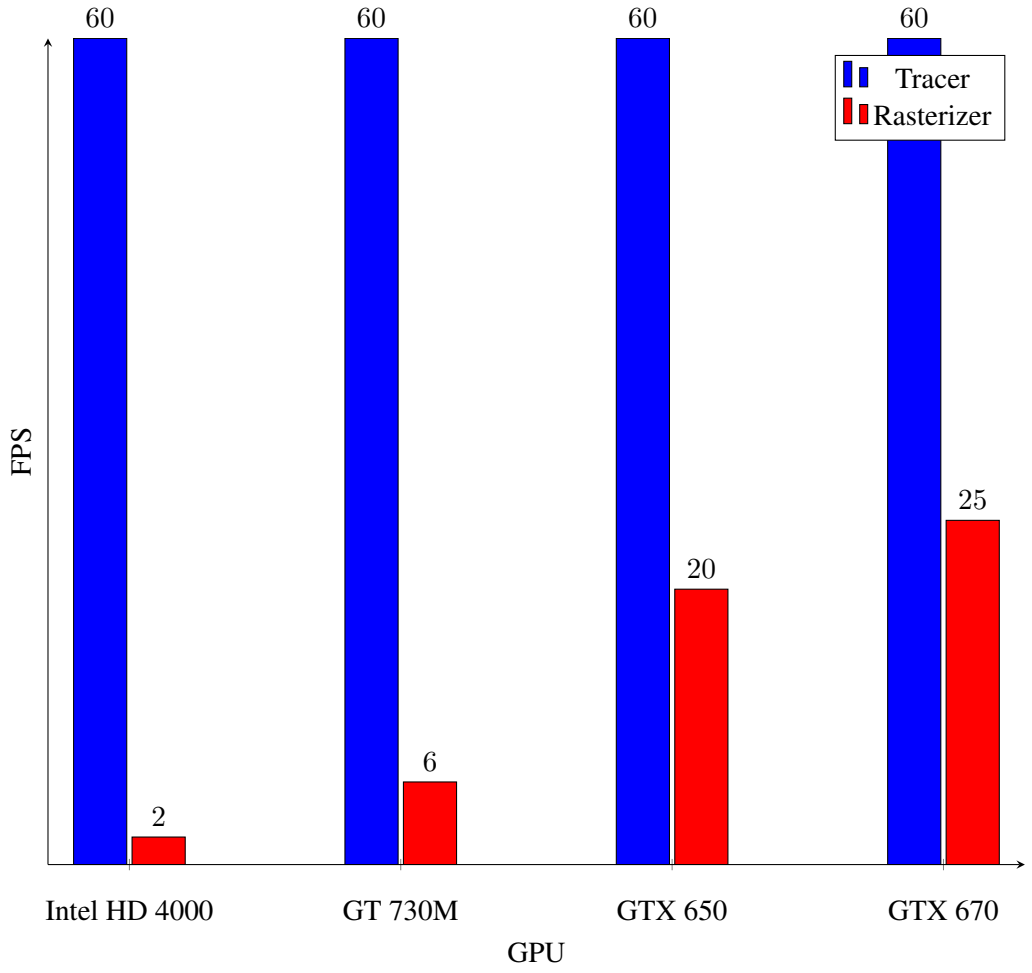

Figure 4.12: Performance comparison

With the given settings, tracer provides better image quality, and as we can see it runs up to 30x faster on hardware with low vertex processing abilities. High end GPUs can handle even higher tracer settings such as AA easily.

5. Conclusion

The proposed ray casting method shows superiority to the mesh rasterization method in image quality, memory consumption and performance. Unlike the mesh rasterization method, its quality is not dependent on the domain size, and it requires a constant precomputation time. Moreover, the ray casting method requires no more hardware features or graphics API support than the mesh method.
From all this we can conclude that ray tracing function defined surface patches is superior to rasterizing function defined surface patches.

BIBLIOGRAPHY

[1] https://www.mathworks.com/products/matlab/.

[2] https://www.wolframalpha.com/.

[3] https://en.wikipedia.org/wiki/Ray_tracing_%28graphics%29.

[4] https://en.wikipedia.org/wiki/Bisection_method.

[5] https://www.opengl.org/.

[6] https://en.wikipedia.org/wiki/Phong_shading.

[7] https://www.opengl.org/wiki/Vertex_Specification.

[8] Tavian Barnes. Fast, branchless ray/bounding box intersections, 2011.

[9] Bjarne Stroustrup. *The C++ Programming Language (4th Edition)*. 2013.

Postupak praćenja zrake na zakrivljenim površinama

Sažetak

Ovaj rad obrađuje metode crtanja funkcijski zadanih zakrivljenih površina. Obrađeni su uobičajeni način crtanja mreže trokuta, te metoda praćenja zrake. Preformanse i karakteristike spomenutih metoda su uspoređene. Navedeni su implementacijski detalji programske potpore koja dolazi uz ovaj rad, te su sve važne implementacijske odluke objašnjene.

Ključne riječi: praćenje zrake, rasterizacija, mreža trokuta, funkcija, bisekcija, pronalaženje korjena

Ray Tracing Surface Patches

Abstract

This paper covers methods for drawing function defined surface patches. The standard mesh based method and a ray casting method are covered. These methods are compared, and their performance and limitations are commented on. Implementation details are covered and detailed explanations of implementation decisions and optimizations are provided.

Keywords: ray tracing, rasterization, mesh, function, bisection, root finding