Ray Tracing on the Cell Broadband Engine

Charles Lohr David Chapman
University of Maryland, Baltimore County

Abstract

We propose an interactive ray tracer for use on the Sony PlayStation 3, using only its CPU, the Cell Broadband Engine for all computations. We created an interactive ray tracing platform that can display primitives, particularly spheres that are static, dynamic, and can be controlled easily via a library. Our output is viewable in HD on the device that is generating the video via HDMI or VGA.

We developed this application by utilizing the massively parallel problem that ray tracing presents by fully utilizing all six SPUs that the PlayStation 3 provides. We have made this ray tracer interactive using many optimizations, parallelizing tasks at many levels. Most of my focus will be on making an intersection engine that can perform combinations of one-rays-to-one-object, many-rays-to-one-object, many-to-many-objects and a frustum-object-collision test.

In this paper, I will focus more heavily on my involvement in this project and on the need for heavy optimization in the area of performing intersection tests.

Keywords:real time, raytracing, CELL processor.

Introduction

Video games have always based themselves around rasterization graphic rendering methods, and non-real-time-methods have almost always based themselves around ray tracing. This is because rasterization methods are generally speaking fast, and ray tracing methods are generally speaking slow, but are able to easily do effects that are extremely difficult to perform with rasterization techniques. There are exceptions to this rule, as there are cases where massive quantities of computing power have been assembled to perform real-time ray tracing for a video game [Schmitter, et al. 2004]. These cases are almost purely academic.

We want to focus on a method that can be performed using regular hardware that over two million Americans have sitting next to their television sets, the PlayStation3. Our method will also work on the platform entirely using free software. Additionally, we can only utilize the processor, as the RSX video processor can only be accessed by a type 1 hypervisor, as a framebuffer.

While our goal is common and will take a fair amount of teamwork at the end to complete, the parts of the engine can be separated easily. We will split the ray tracer up into the following parts: the user layer, the mechanical layer, the ray-driver layer, the "datastructure," the intersector, and the shader. I plan to focus on the mechanical layer, the user layer and the intersector, as well as touch upon the idea of a shader. I will mostly cover the parts that I will work on.

The mechanical layer is what enables input from a user's program, sets up the environment for ray tracing, handles controlling the video output and user input devices as well as handling management of the six available SPUs. The mechanical layer will also handle all DMA (Direct Memory Access) between each SPU and main memory along with the code to dynamically move the output of the ray tracer into the framebuffer for display on a connected VGA monitor/HDTV.

The user layer is how a user of this "library" will interface to it. The final version of this part is designed as an executable, where the user's code is statically linked in. Once linked in, every frame, the user's code is called with all necessary frame information. In our current example, the user's application may choose to or choose not to modify default parameters of the system, but will be required to perform actions like position the primitives. For our code, we are hard-coding the system to support exactly sixteen spheres.

Much of my effort on this project was on the intersector. The intersector is much less comprehensive than a collision engine, it simply does ray-primitive collision. For this project, hand-writing a basic ray-sphere and ray-triangle intersection is not sufficient. I experimented with different ordering, dealing with multiple simultaneous and sphere and triangle collisions at once. I coded multiple rays to single primitive, as well as one ray to one primitive and one frustum to one primitive. Additionally, the code to perform sixteen rays-to-sixteen primitives is included but not heavily optimized. After coding a series of different functions to handle the collisions, I verified their outputs and clocked them in different situations to see how fast they behaved in reality on the PlayStation 3's SPUs. Now complete, I have a library that Dave can pick the best of to make the ray tracer run as fast as possible.

Related Work

The PlayStation 3 uses a very odd processor, the IBM CELL processor, an asymmetric, 8 core processor. It has a lot of power but takes a number of considerations in order to utilize this power. All of the tactics that I utilized with respect to this project are described in this paper.

Chow, Fossom and Brokenshire [2006] were among the of the first to effectively show how processes can be parallelized, even down through the instruction level on the CELL processor. While their work was with performing FFTs on the CELL processor,

they helped show specific vectors a programmer could look when attempting to optimize a program to run on that architecture.

Previous work on the PlayStation 3 with regards to real time ray tracing only covered static scenes of a very basic and direct nature [Minor, Gordon 2005], or were limited to very specialized types of scenes, such as height maps [Benthin 2006]. Our goal is to have an engine that can handle more than just triangle meshes as well as being able to handle objects moving around.

Because of some of the newer innovations in Coherent Grid Traversal [Wald 2006], we can make use of performing intersection tests on packets of rays simultaneously. Between this and the already massively parallel structure that the cell processor makes natural, we are able to ray trace very simple dynamic scenes at an interactive frame rate.

Background and Outline of Implementation

The user layer and mechanical level of this process are somewhat direct and do not require much research. Of particular lack of academic interest is the user layer that will be ignored for the rest of the paper.

The mechanical layer is of a fair amount of interest. When using Linux on the PlayStation3, the video card is accessed under a hypervisor. Conveniently, the way in which the screen is accessed is memory mapped video, also much to our benefit, this memory is accessible directly from the Synergistic Processing Units (or SPUs,) when using DMA. This allows us to have the output of the highly optimized code running on the SPUs to be sent directly to the framebuffer, without having to make either the PowerPC Processing Units, or PPUs or the SPUs spend cycles copying to and from the local 256KB store. By freeing up the PPU, an application such as a video game can spend ample CPU time formulating the scenes instead of worrying about rendering them.

All of the code that we are focusing on and trying to make extremely fast will be written in assembly. The reason for this is that we intend to use open source compilers for this project, keeping everything open source and non-proprietary. Currently, the GCC compiler does not optimize very heavily, or effectively for the CELL architecture, and we have been able to gain up to a 4:1 speed up when we write assembly code, compared to compiling optimized (O3) C code to do the same task.

The core of my research on this project is based on the intersection of rays, packets of rays, and frustums with an object, or numbers of objects. I first intersected one object with one ray and then rewrote the code to intersect 16 rays to a single object. I spent much effort finding ways of moving instructions and data around to achieve high performance within the constraints of the SPU pipeline. I found major speed increases when intersecting multiple rays to a single object, and less noticeable speed increases when intersecting multiple objects, probably due to less effort in the optimization.

The overall plan for implementation is that we have an application running on the PPU part of the processor. This application builds the scene as well as all basic data structures that are used by Dave's algorithm. It then sets up all of the variables for performing a Whitted-style ray tracer [Whitted 1980] (minus shading). From there, it generates work units and starts to determine a way of dividing the work between each of the six SPUs. It may split the work units even further, should we find this to be beneficial. As of now, we are simply dividing the screen into six portions, one per SPU. Once all of the work units are ready, they are 'shipped off' to the individual SPUs. The SPUs perform the work necessary for these work packets, and upon completion of a group of scan lines, it will use DMA to send the data back to the framebuffer.

The work units are composed of the scene data, any additional structures that are necessary, the viewing frustum, what pixels the work unit is responsible for, and the location in the hypervisor to output the completed ray data.

When testing the code, most of the information regarding each ray is stored in the SPU's registers, and most of the information regarding the objects that the rays are hitting are stored within the SPU's local memory store. By keeping this uniform, I will be able to stay focused only on changing the two variables that I am interested in changing: number of rays and number of objects.

Mechanical Outline

In general, when writing code to run on the cell processor, there is generally a host application that is run on the CELL's PPU. This host application then can start tasks on the SPUs. In general, the PPUs are particularly well suited to general purpose computing. They are able to perform loops and if statements with little overhead, as well as talking directly to the host operating system, for our project this is Linux. The SPUs conversely are suited to streamlined, SIMD tasks, they are extremely powerful floating point processors, however they are very poor at out-of-order code execution and not share ram directly with the main memory that the PPU uses. All communication between the SPUs and main memory are through DMA so that none of the processors have to spend clock cycles moving data around. See Figure 1.

For our project, the main core of our ray tracer is located on the PPU. This is what generates the geometry and viewpoint for the scene. The PPU app then prepares our ray trace engine to run on the six SPUs. The PPU app splits the workload between the SPUs evenly. It generates a “work unit” containing information such as the geometry in the scene, the camera position, slice of image that the SPU should compute and the absolute location in ram where the SPU should output the completed pixel data See Figure 2.

Each of the six SPU application are started simultaneously and upon start, they use DMA to get the work unit onto the local cache. Once this is accomplished, Dave's algorithm will be run. For every group of rays, he will determine what objects the ray(s) could potentially hit. As it stands now, and as described in my paper, only the testing code is used to ray trace.

We split the work up into 16 rays at once, using 4x4 packets. Since our output resolution is 1280x720, this means each group of four line will contain 320 packets. Once all 320 packets are done, and the outcome of their information is completely written to a 4-line buffer in the SPUs local cache, the entire group of lines is then copied (via DMA) to the hypervised video memory.

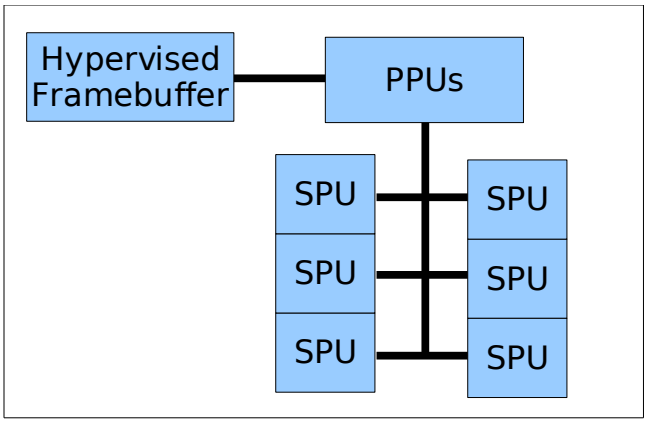

Figure 1: General Layout of the PS3 System

Because four lines of data exceed the maximum length of data that can be transferred in one DMA packet, the data is split in half and sent as two different DMA packets. The DMA requests are non-blocking, so having two requests is not an issue.

Because there are six SPUs, and we are splitting the workload evenly, each SPU ends up only needing to compute 120 lines, or just 30 of the block-lines. See Figure 3.

During the execution of the program, in order to actually utilize the frame buffer effectively, VSync must be enabled. Without VSync, the video output behaves unpredictably. All frame rates, unless otherwise noted are measured with VSync on, using 1280x720 on an HDMI output device.

Intersector Outline

The intersector code is the most time-critical code. While the primary intersector intersects sixteen rays simultaneously, when rendering only one sphere, if unoptimized, it must execute 9,600 times on each SPU to generate a single image. In a general scene, this code is expected to be called hundreds of thousands of times per second using Dave's method. In our tests, when rendering sixteen spheres, this code is executed 153,600 times per frame on a single SPU.

The SPU's assembly language functions mostly on two to three registers at a time. The SPU contains a total of 128 registers. Each register contains four single-precision numbers. In general, most SPU asm calls work best when operating with two registers, and outputting to a third. For example, the FA function will take two registers, and component-wise add them into a third register. This means that whenever possible, we should be multiplying, dividing, or performing other operations in parallel on registers.

My main focus on this code is to figure out the best way to pack data, and perform the SIMD instructions that the SPU performs best to get as much work as possible done. For sphere collisions, on a per-ray basis, we tried using a cross product based system to determine intersection, but abandoned it for a quadratic equation method. We ended up using the following function to determine if a ray intersected a sphere. See Figure 3.

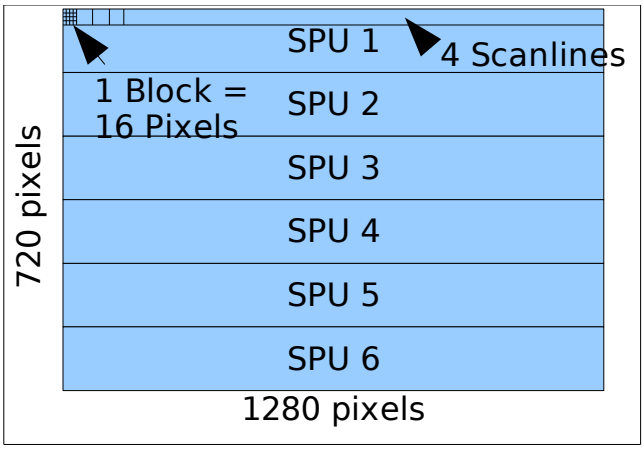

Figure 2: Work Flow Layout

$$a = D_x^2 + D_y^2 + D_z^2$$
$$b = 2D_x P_x + 2D_y P_y + 2D_z P_z$$
$$c = P_x^2 + P_y^2 + P_z^2$$
$$T = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Figure 3: Equations for Ray-Sphere Intsection

Where D is the direction of the vector, P is the location of the sphere minus the location of the camera and T is the distance of the intersection from the camera in the D direction. Note that this algorithm demands that the values in D be normalized. In the event that they're normalized, a will always be equal to one. Also note that for all rays, because the value of c will be equal, c only needs to be computed once for all sixteen rays.

Some of the major hurdles with the SPU are both the poor branching and the SIMD code. All of the PlayStation 3's registers contain four floating point values. Most floating point operations act on two registers with a third as an output. For example, the *fa $rt, $ra, $rb* function adds registers $ra and $rb, and puts the sum in $rt. This goes component wise through the registers. This does not help when many values need to be added sequentially, such as adding all of the components together in a register.

One of the major optimizations that I use many times in my algorithm is to transpose a matrix of four registers. This is particularly useful when trying to add all of the components of a vector, such as what is necessary for finding the magnitude of a vector or computing something like b. This can be performed using the following code, see Figure 4.

We only need the first three components in most situations in our algorithms, so only reshuffling the first three components is all that is necessary. The reason that it makes so much sense to order the data in the way shown in the second block is that it allows one to sum the components in a, b, c and d very quickly, in fact, in exactly three *fa* instructions. The output can be a single register which component-wise represents the sums of the components in a-d.

a	<aX>	<aY>	<aZ>	<aW>
b	<bX>	<bY>	<bZ>	<bW>
c	<cX>	<cY>	<cZ>	<cW>
d	<dX>	<dY>	<dZ>	<dW>

Is transformed into:

ra	<aX>	<bX>	<cX>	<dX>
rb	<aY>	<bY>	<cY>	<dY>
rc	<aZ>	<bZ>	<cZ>	<dZ>

Using this following code:

```
  shufb ra, a, b, shuffleAE00;
  shufb rb, c, d, shuffle00AE;
  shufb rc, a, b, shuffleBF00;
  shufb rd, c, d, shuffle00BF;
  shufb re, a, b, shuffleCG00;
  shufb rf, c, d, shuffle00CG;
  <execute unrelated instruction here>
  or ra, ra, rb;
  or rb, rc, rd;
  or rc, re, rf;
```

Figure 4: Quick transposition for float re-ordering

For the duration of our sixteen-to-one sphere function, we can hold all 16 *b* values in only four registers. We can also perform all functions, like the squaring, subtracting 4ac, square rooting, etc. on four rays simultaneously.

Because my setup is in the form of a whitted-style ray tracer, and our algorithm works much faster if D has been normalized, we have to take the extra step to normalize all rays before continuing. This normalization benefits from the optimizations we're performing here.

There were a number of pipeline optimizations that I produced. One of the major benefits of testing four rows of four pixels each is that first of all, we can do the optimizations I described above. But additionally, we can separate groups of code naturally and in the pipeline. For instance, if we wanted to calculate E = AB + CD, and we had sixteen of each variable, we organized the code to be similar what can be seen in Figure 5.

This sort of optimization was performed many times over in the sphere-ray intersection algorithm. I believe we have taken the embarrassingly parallel problem of ray tracing and broken it down to two more levels of ridiculous parallelism, effectively making it even more embarrassing.

The frustum algorithm benefited most from the parallelism at the register level, and was able to be condensed down to 21 instructions to test a sphere against four clipping planes, using the equation shown in Figure 6. N_{Plane} is the plane's normal, S_{Center} is the relative center of the sphere WRT the camera.

I did not write any working triangle intersection algorithms. While the framework exists for a shader, I decided to comment out the jump and simply shade the color as distance from the eye.

```
  fm tmp1, A1, B1
  fm tmp2, A2, B2
  fm tmp3, A3, B3
  fm tmp4, A4, B4
  fma E1, C1, D1, tmp1
  fma E2, C2, D2, tmp2
  fma E3, C3, D3, tmp3
  fma E4, C4, D4, tmp4

Offers more than a 3:1 speed increase over

  fm tmp1, A1, B1
  fma E1, C1, D1, tmp1
  fm tmp2, A2, B2
  fma E2, C2, D2, tmp2
  fm tmp3, A3, B3
  fma E3, C3, D3, tmp3
  fm tmp4, A4, B4
  fma E4, C4, D4, tmp4
```

Figure 5: Ordering of instructions matters.

$$Visible = (N_{Plane} \cdot S_{Position}) < Radius$$

Figure 6: Equation for frustum checking

Results

We had extremely good results, Figure 7 was rendered at 1280x720 pixels using my algorithm. No early clipping was perfomed, and every ray was tested with all sixteen spheres. When operating V Synced, it received thirty frames per second.

Originally, my test code was written in C, rendering only one sphere. I received 15 V Syned frames per second when running that code. I did not implement any systems to do early fail detection using the frustum culling algorithm, or the algorithm described earlier in this paper. These rates are brute force.

Also, when testing between partially optimized and the final product, we found that the time it took to render sixty frames went from 5.9 seconds to just under five seconds, allowing us to receive our twelve FPS. When we utilized all six SPUs, our render time went to two (vsynced) seconds (30 FPS). When we stopped vsyncing and writing to screen, our speed went up to 52 FPS. Additionally, we rendered 13 spheres at higher rates (Figure 9.)

We found that the overhead of getting the work units to the SPUs and getting the data back is surprisingly small, considering the software limitations that the PlayStation3 has without special licensing.

The end user application is somewhat less than what was expected. But it does allow a user to have code in an external file, game.c. This file executes on the PPU, so it has direct access to a general purpose processor and the host operating system.

Additionally, when utilizing our system, the PPU was left largely free to perform any code necessary related to game play. We found that during our system's execution, the PPUs were over 95% free. See Figure 8.

We have produced a framework and program that allows other programs to very simply render up to sixteen spheres, at exactly

twelve FPS on an HDTV using a PlayStation3. Possible games that could be written using this framework include billiards and possibly a screen saver.

```
CPU0  :  0.3% us,  8.0% sy,  0.0% ni, 91.7% id
CPU1  :  0.3% us,  0.0% sy,  0.0% ni, 99.7% id
```

Figure 8: Top usage per PPU during application run.

Future Work

In the future, I plan to continue this work and attempt to add programmable shading code, to allow objects to specify a type of shading on a per-object basis, allowing the programmer of the game to write a custom shader for every object on the screen. In the event that I do this, the shaders will still have to be written in assembly, but they should be fairly easy to code, considering the large number of registers and simplicity of the instructions provided on the SPU.

I would also like to attempt to find a better way of splitting the workload between each of the SPUs. As of now, splitting it evenly across all six SPUs can result in one SPU finishing late, thus leaving all the other processes idle, while taking up precious time in the frame.

No attention was paid to even/odd instructions on the SPU that could potentially speed up the ray tracer. I may investigate that feature of the SPUs to provide another speed boost to the algorithms.

Additionally, I would like to take my part of this project, and make it independent from Dave's, and implement a ray tracer that supports reflaction. While with the method we are currently investigating to down on the collision tests would not allow this, we could make a ray tracer that does not utilize this and simply runs slower or with less objects, with more advanced effects.

Instead of providing full triangle support, we would like to investigate the ability to have specialized functions. This would allow me to specifically make a game like a billiards game with a table.

Schmittler, J., Phol,D., Dahmen, T. 2004. Realtime Ray Tracing for Current and Future Games.

Chow, Alex C.; Fossum, Gordon C.; Brookenshire, Daniel A. 2005. A Programming Example: Large FFT on the Cell Broadband Engine.

Minor, Barry; Frossum, Gordon; To, Van. 2005. Terrain Rendering Engine: Cell Broadband Engine Optimized Real-time Ray caster.

Benthin, C.. Wald, I,., Scherbaum, M.. Fredrich, H. 2006. Ray Tracing on the Cell Processor;

Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, Stepheen G. Parker. 2006 Ray Tracing Animated Scenes using Coherent Grid Traversal.

Whitted, T. 1980 An improved illumination model for shaded display. Communications of the ACM 23, 6, 343-349

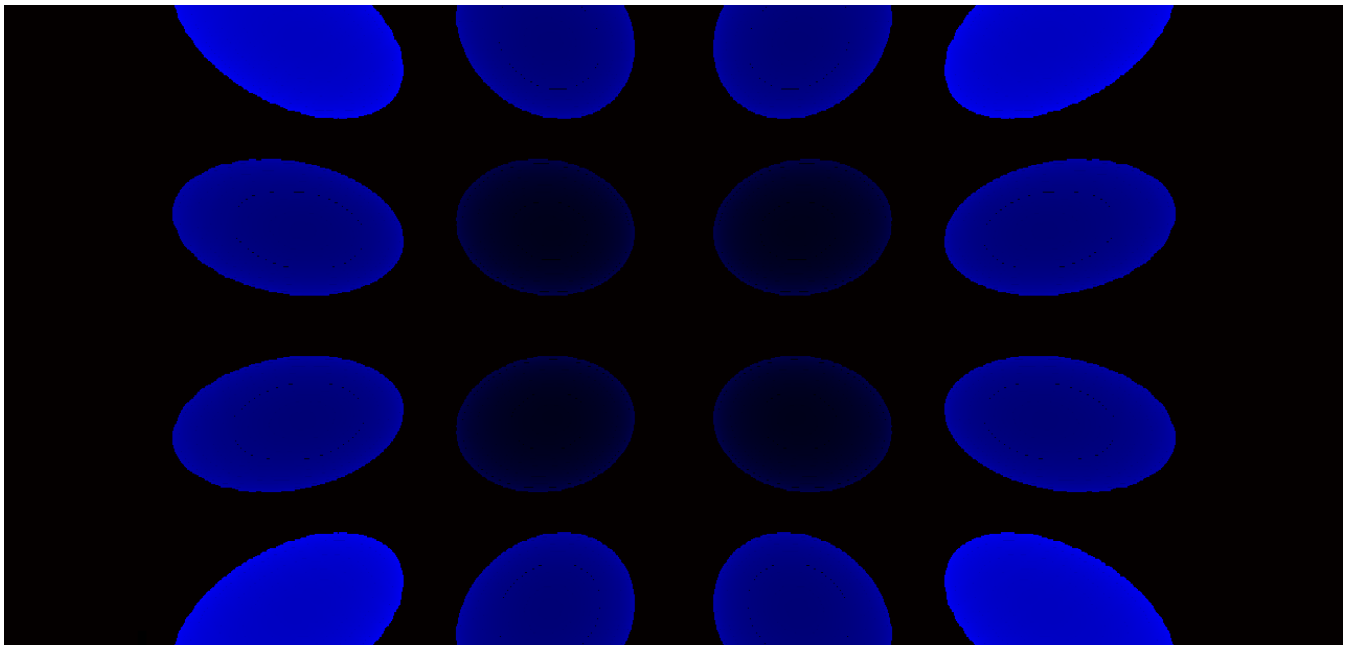

Figure 7: Sixteen spheres rendered at 30 FPS (Vsynced), 52 FPS without vsync.

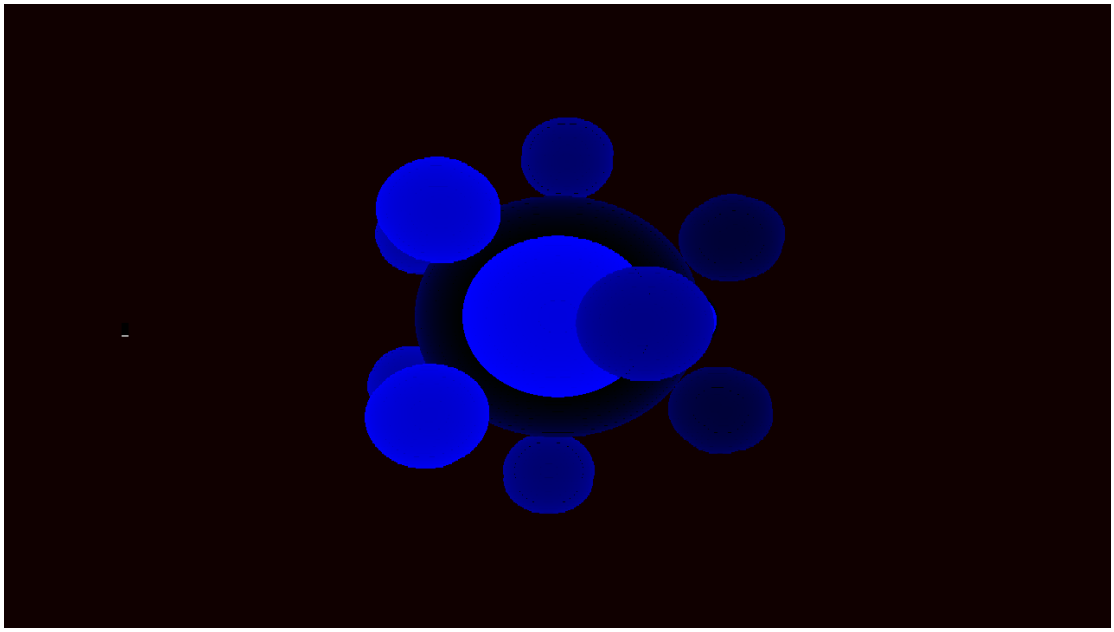

Figure 10: Thirteen spheres being rendered at 60 FPS,
when uncapped, rendered at 64 FPS.