COSI 155B - PA3

Michael Partridge, Kahlil Oppenheimer, Eden Zik

For our project, we decided to implement two small features (anti-aliasing and multithreading) and three main features (Depth of Field camera, arbitrary quadric surfaces, and ray-tracing of animated scenes). We were able to fully implement all three features we set out to implement originally. Descriptions of each feature are included below.

Anti-Aliasing

Each of our cameras supports oversampling within a +/- 0.5 pixel jitter for antialiasing. The number of samples taken is a configurable field. We simply take the specified number of samples randomly distributed in the space for a particular pixel, trace a ray through each, and average the returned colors. The outcome is that the image looks less jagged, and quick changes of color are smoothed out.

Multithreading

We realized early on that ray-tracing is very easily parallelizable across multiple threads. Each thread can simply execute the tracing of a single ray, since that calculation is independent of the calculations of all other rays being traced.

Because this task is CPU-bound, rather than I/O bound, we knew that parallelizing this task across too many threads would add too much overhead cost. However, because many CPUs have hyperthreading enabled and CPU scheduling algorithms don't guarantee that all of our threads will run on separate cores, we suspected that there might be an advantage to using more threads than the number of cores. We experimented and found the best performance when using 2-3 threads per physical cores of the machine the program is run on.

Depth of Field

A depth of field camera with user-specified focal distance and aperture size is implemented. The algorithm simulates a larger than pinhole opening in the camera through which light is allowed to enter and expose the film. This has the effect of causing a single pixel on the camera film to represent a small area in the scene instead of a single point. By casting many such rays and averaging them (in the same manner

as oversampling for antialiasing) we can produce a blurred effect for objects outside of the focal 'plane' similar to a camera's depth of field effect.

The algorithm is relatively simple: first a ray is cast as in a standard camera through a pixel in the film. However, instead of checking for objects intersecting that ray, the depth of field camera finds the point at which that ray is a focal length away. This is the focal point of that ray. The camera then picks a random (uniform over a square aperture) point on the aperture for the origin of the ray. A new ray from that origin point through the focal point is then traced and mapped to the original pixel. This ray will be very close to the original intersection point in the scene for intersections with objects very close to the focal point, but as the aperture gets larger and distance from the focal length gets larger, the rays will diverge and a blurred effect is produced when many are averaged.

The aperture is a square centered on the camera in the XY plane. The user specified aperture setting is inversely proportional to the size of the aperture on the camera similar to f-stop settings on a real camera. The diameter of the aperture in standardized (-1 to 1) units is 1/aperture setting. Similar to a real camera, high 2-digit aperture settings (f-stop settings) produce a scene that is nearly all in crisp focus and low aperture settings (single digit f-stop) produce a scene with most out of focus except for the object centered at the focal length of the camera.

Quadric Surfaces

We extended our ray tracer to support tracing arbitrary quadric surfaces of two-dimensions situated in Euclidian space. A quadric surface is any surface of the general form Ax^2 + By^2 + Cz^2 + Dyz + Exz + Fxy + Gx + Hy + Iz + J = 0, where A, B, C, D, E, F, G, H, I, and J are real valued constants, and x, y, z are variables.

Many shapes we had already defined are actually also quadric surfaces, such as spheres and cylinders. However, we support any arbitrary quadric surface, which include ellipsoids, elliptic paraboloids, hyperbolic paraboloids, elliptic hyperboloids of one or two sheets, etc.

To accomplish this, we first realized that all we needed to do to ray trace a surface is to describe how a ray intersects with it. To describe the intersection, we just need to calculate the point of intersection and the normal to the surface at that point.

First, to calculate the point of intersection, we realize that we already have a parametrized equation of our ray. We can plug each x, y, and z component of our parametrized equation into the x, y, and z variables in our general equation for quadric

surfaces described above. Using some algebra, we end up with a quadratic equation in terms of our parameter of our ray equation. Now, we just solve that quadratic equation to find the the earliest, positive parameter value, and plug it back into our ray equation to find the point of intersection.

Finding the normal to the surface at the intersection is straightforward once we have the point of intersection. All we need to do is take the gradient of our surface, then plug our intersection point values into each component of our gradient to get a vector moving in the direction of the gradient (i.e. normal to the surface) at the intersection point.

With the intersection point and the normal at the intersection, we have sufficiently described our ray hit enough to ray trace it. The only caveat is that we were not able to generate a generalizable solution for coming up with texture coordinates at any arbitrary quadric surface ray intersection. Thus, our quadric surfaces do not support custom textures.

Lastly, we added the ability to bound the maximum/minimum x, y, and z values for any quadric surface. This lets us accomplish tasks we approached differently before (like bounded cylinders). It also lets us define cool surfaces, like hourglasses formed from bounded elliptic hyperboloids.

Animations

We extended the Ray Tracer with the ability to specify parametrized transforms, and hence shift an object's position or rotation independent of its coordinates. These transforms were the basis for animations, done by iterative rendering of a transforms over an initial frame to produce the next.

Each scene consisted of a set of objects with initial position. These definitions were the basis for the initial frame, at which point an arbitrary set of functions (transforms, object addition,lighting change, etc.) are applied to produce the next frame in the sequence. The frame produced at each iteration is the result of the function set applied on the previous frame.

All frames produced in this manner are then written in sequence as a PNG encoding to a GIF file, with a header defining the frame rate. The stitched images can then be viewed in a web browser or other platforms supporting the GIF format.

The API for constructing an animation is based on the one for constructing a single scene - and employs an Animator callback which supports modifying the state of the

current scene to progress it to the next frame. For example, in a scene consisting of a sphere at position (0,0,0), the main rendering process would render the scene and produce an initial GIF file consisting of an image depicting the sphere at this initial position. After rendering the first scene is finished, the callback Animator is evaluated to produce the next frame - which is the same scene with a modified internal state where the sphere is at position (1,0,0) - as a result of a transform along the X-axis.

We realized animations look best at 24 frames per second, and adjusted our parameters accordingly. However, it is difficult to debug a scene that is more than a second long without waiting a prohibitively long amount of time for all of it to render. We corrected this using an easily adjustable parameter, producing choppier but overall isomorphic scenes at lower frame rates.

The GifSequenceWriter.java is used under a Creative Commons License (attributed in the file) and is not our own original work. All other aspects of the animation are original works of this group.

Demos

We included several demos to showcase the features we implemented. A short description of each is included below.

Three demos are used to show **depth of field effects**:

DepthOfFieldStill.png

A still image contains various objects in the background and a green sphere in the foreground, all out of focus. This camera is using a large aperture, which creates a short focal length. A single textured cylinder in the middleground is in sharp focus, clearly behind the green sphere and in front of the other objects because of the various overlaps.

Walking.gif

A moving image models a camera walking through a small structure composed of columns and a roof. The depth of field is narrow and fixed which causes columns on either side to briefly come into focus and then go out just as they leave the frame. A blurred horizon and clouds in the sky appear far off outside the focal distance.

Moving-shadows.gif

One demo also uses a depth of field camera, however its effects are much less pronounced. It features a fixed camera observing the same structure as in the depth of

field demo while the sun travels across the sky simulating a day in the northern hemisphere. Shadows swing around columns and the scene brightens and darkens slightly to simulate dawn and dusk.

Three demos are used to show **Quadric Surfaces effect:**

ExpandingHyperboloid.gif

An animation containing a hyperboloid expanding, contracting, and rotating. The hyperboloid is rotated about the x and y axes as it continually expands/contracts. This demo demonstrates the power and versatility of arbitrarily bounded quadratic surfaces that can be translated, rotated, etc.

QuadricStill.png

This still image contains a bounded hyperboloid of one particular orientation situated in front of two other unbounded (at least in one direction) hyperboloids behind it. These still image demonstrates the power of arbitrary bounds, in that bounding hyperboloids in arbitrary places can yield pretty interesting visual effects. Additionally, we see all the expected ray-tracing features like shadows, reflections, and so forth working as they should.

One demo is used to show **Transforms and Animation:**

Basketball.gif

An animation of a basketball shooting through a hoop. This image demonstrates parabolic transformations, and as such the basketball is able to land in the hoop while the camera shifts to capture said moment. We still see the reflection of the basketball on the wooden floor (a texture) while it goes up in the air.