Computer Graphics: The Path of Light report

s1763241

1 Introduction

While building the ray tracer I closely followed the steps as described in the assignment's workflow description and marking scheme points. Following that ordering of tasks this report is structured in the same order as the components listed in the marking scheme. We start at the fundamentals such as Camera and Ray functionalities, going through the shapes and their materials implementation, shadow and reflections casting, finishing with Bounding Volume Hierarchy implementation and distributed ray casting (partially). Unfortunately due to time constraints only AreaLight part of the distributed ray tracing has been implemented and also just partially.

The very first task is to complete the parser to make sure that all of the scene components can be created, initialized and destroyed correctly. This was done by adding print statements to the constructors of all shapes, cameras and lights. All of these classes also have empty constructors, but when completing this task I decided to already use constructors with all their parameters they have in the json file used to describe the scene. With this approach it was slightly faster to add new functionalities as classes already had all their parameters parsed, saved and ready to use. The specific parameters of some classes were however often adjusted, added or removed, as the new tasks were done and it was clearer what needs to access which parameters. Creation of the subclasses (e.g. Pinhole is a subclass of Camera) was done using the class factories, so for Camera there is Camera::createCamera function, for lights this is LightSource::createLightSource, for shapes Shape::createShape, and so on. Each of these takes its respective json description taken from the main json input file and returns a corresponding instance of the subclass.

Once this task was done, next step was to start implementing the basic ray tracer.

2 Basic Ray tracer

This section describes all capabilities of the basic ray tracing algorithm, starting by the defining how rays are being casted into the scene, following by how they are created by the pinhole camera and the resulting image details. After that we move to shapes such as intersections calculation and finish with materials and shading implementation using Blinn-Phong shading.

The light that is being used is defined by PointLight subclass of the LightSources class. PointLight is defined by position, id and is parameters that are used in the shading and thus PointLight only contains getter functions to return these parameters so that they can be used by the scene's castRay function.

2.1 Ray casting

Ray casting is implemented by the RayTracer class that has two functions: render() and tonemap(). Former is used to create the resulting image by looping over each pixel in each row and column, calling a camera to get the ray for current pixel and then sending the ray to the scene using scene's castRay function to get the corresponding color of this pixel. Once this is done and all pixels' colors are filled, the tonemap function is called which first find the maximum color value over all R, G and B colors in all pixels, then divides them all by this maximum value found to normalize the RGB channels to all be in the same 0 to 1 range, and then finally multiplies them by 256 to get into the 0-255 range that we are used to see when it comes to colors.

As mentioned above, the castRay function defined in the Scene class takes a ray as a parameter and returns a color of the pixel that the ray is being called for. This color can be either background color of the scene if the ray misses all shapes, or can have some other color based on where the ray landed. CastRay function is a recursive

function, when the ray parameter originates from the camera, called a Primary ray, hits any object, another call of castRay is performed to check for any reflections (Secondary ray), but more about this later.

2.2 Pinhole camera

Pinhole camera is defined by several parameters taken from the json file, once these are processed, the main parameter we need to create in order to be able to cast primary rays is matrix called cameraToWorld matrix. This 4x4 matrix is used to transform the pixel coordinates of the rendered image to the world coordinates - our 3D scene coordinates, and then these are used to create rays from the camera by setting the ray origin to be the camera position and ray direction being the direction to the image pixel in world coordinates, normalized to be a unit vector.

CameraToWorld matrix is calculated using the lookat and up parameters read from the json description of the camera. Normalized lookat is defining the z axis of the transformation, the x axis are then calculated as a cross product of x azis and normalized up vector. Finally y axis is defined as a cross product of x and y axis. The cameraToWorld matrix is the homogenous matrix defining rotation and translation so the first 3x3 upper left corner is set to x, y, z axis rotation (x at first row, y at second and z at third) and the last column on this vector is defined as translation - x, y and z position of the camera in the scene.

When creating the primary ray from camera, the origin is always the same - camera position in the scene, but the direction of the ray depends on which pixel in the screen we are looking at. We first normalize the pixel coordinates (raster space) to be in 0 to 1 range. Then we shift them in such a way that the middle of the screen (the image) is at (0,0) coordinate and also take the FOV into account. The transformation from raster to camera space was inspired by the tutorial available online here. Once we have the pixels defined in the camera space we use them as x and y coordinates of the ray direction, with z equal to 1, and multiply the normalized value of this vector with the cameraToWorld matrix to get vector in the world space coordinates.

It is difficult to demonstrate and prove that ray casting and pinhole camera are working correctly as all we can see is one colored image since we don't have ray hits yet. Moreover, even after implemented these two steps, I often went back to them either because axis were reversed, or camera position wasn't set up properly and everything was closer in the resulting image. That's why this report contains only images illustrating progress from this point on. Without these two fundamental tasks, everything else (almost) after this point would not work as intended (or not work at all).

Now we have defined how the rays are created using our pinhole camera and are able to cast the rays to the scene. Now the next subsection describes how the shapes are defined, and most importantly, how we know if a ray hits a shape and what needs to happens when that's true.

2.3 Shapes

All shapes used in the scene (so excluding lights and cameras) are defined by an class Shape which is then further defined by subclasses Sphere, Triangle, Plane, TriMesh and BVH. The Shape class keeps the material parameter (pointer) because every subclass uses it, and also defines functions getMaterialColor and getAmbientColor. All other non static functions such as intersect function are set to virtual and defined by the subclasses. To create a concrete Shape instance such as Triangle, a class factory Shape::createShape function exists and is used in the scene to create the shapes based on the parameters passed.

Each subclass of Shape needs to implement intersect(ray) function which given a ray finds if the ray intersects the shape and returns a Hit structure that contains parameters such as hit point, hit t which is the distance from ray origin along ray direction showing how far away from the ray origin the hit occurred, hit normal that is the normal of shape's hit point and reference to the shape itself that was hit stored as a pointer.

Resulting image of the provided example scene with all objects rendered with just their base colors being the diffusecolor value from json file can be seen in figure 1. This include checking which hit point is closest to the ray source when multiple shapes are intersected by the ray, and only choosing the closest shape hit. This is clear from the 2nd image where we see that part of the sphere is hidden behind the plane. To be able to show this image

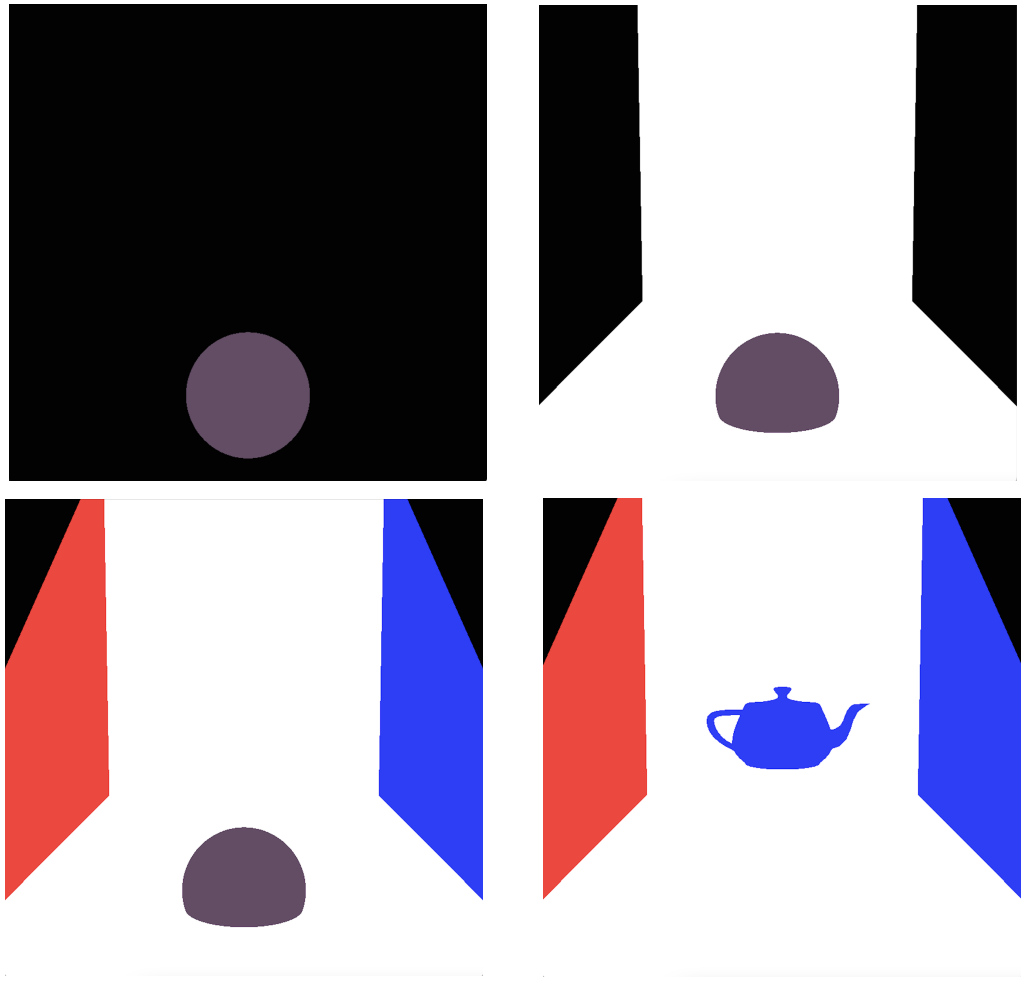

Figure 1: Basic renderer rendering Shapes based on their constant color defined by json input. Looks at closest hit color only.

there is also tonemapping implemented with simply just mutliplies the value in the pixelbuffer with 255 to move from 0-1 range to 0-255 that we are used to.

2.3.1 Sphere

Sphere is defined by a center and radius, these are stored as parameters and used when calculating intersects function (and more). To calculate if the ray intersect the sphere we need to solve an equation of t - distance from rays origin along the ray's direction and see if this point is on or in the sphere itself. To do this we solve a quadratic equation as described in the github article that was our solution inspired by. We find the t parameter and get the smaller one because we want the closest point hit to the camera and use these details to create the Hit structure carrying the t we found, point of hit calculated as $rayOrigin + t * rayDirection$ and hit normal which is normalized vector from center of the sphere to the hit point on the sphere. Since this function can also send negative t, meaning hit occurred behind the ray origin, the castRay function in the scene always analyses only hits that are in front of the camera, meaning the t is positive.

To test the sphere intersection, castRay function in the scene class returns a diffusecolor parameter from input file of the sphere whenever it gets positive t, otherwise returns a background color. Result of this is shown on the figure 1 in the first image (from left).

2.3.2 Plane

The next implemented Shape subclass was the Plane, a bounded plane defined by 4 points that are stored the parameters of this class. Similarly to the the sphere intersection calculation, we need to find the distance from ray origin to the point on the plane where the ray hits it, to do this we again need need to solve the equation of t to find where (and if) the plane and ray meet. Math of calculating t is inspired by the Wikipedia's line-plane intersection page. Since this corresponds to an infinite plane, which is not true in our case, we need to further test if the point we found ($rayOrigin + t * rayDirection$) is inside plane's bounds. To see if the point is inside we perform a test called Inside-Outside test where we first define vectors representing plane edges, for example the first edge is a vector from $v0$ point to $v1$ point of the plane, and then we check the hit point found against each of these edges, this approach was inspired by an inside-outside testing explained in here which is a guide for triangle intersection but it is the same for the planar quad, we just have 1 more edge to check.

Resulting rendered planes can be seen in the 2nd image in figure 1, 1st row. We can see that now when we have more shapes we also had to add a check to castRay function to see if the hit's t parameter is closest one found so far, and only return the color of the closest hit. This results in bottom of the sphere to be hidden and rendered because it is behind (or more like below since that should be floor) the bottom plane.

2.3.3 Triangle

Triangle shape is defined by 3 points - its corners in the world space. These are thus the only parameters that triangle keeps. They are stored as pointers because some triangles might share the same points (e.g. vertices in TriMesh shape). The triangle intersection calculation is almost the same as plane's intersection calculation, the only difference is that we check only if the point found to be a point where the two equations of ray and triangle meet is actually within the bounds of the triangle defined by 3 edges, not 4 like we did in Plane intersection. We can use this logic because triangle is flat so we can first assume that it is just an infinite plane and see if ray is going to ever hit it, and then when that's true ($normal \cdot rayDirection! = 0$) we can find t and perform Inside-Outside test as in the planar quad intersection.

With added triangles and their respective colors we can see them being rendered with all previous shapes in the 3rd image on the figure 1.

2.3.4 TriMesh

Triangle mesh shape is defined by a vector (C++ vector) of triangles, each corresponding to one face of the mesh. TriMeshes are defined by a .ply files that the path to is stored in the json description of the mesh. Reason is that defining the mesh by hand and storing all vertices and faces descriptions in the json file seems wrong given that a mesh can easily have thousands and thousands of vertices resulting in a messy and hard to read json file which we need to preview and often change when working on the scene. Moreover, implementing it once the .ply reading is implemented should be straightforward.

TriMesh constructor takes few parameters: center defining where the trimesh should be placed in the scene, path to the .ply file describing triangle faces of this mesh, size scale to scale the size of this mesh up or down and a boolean indicating uses of BVH (more about it later). To create the mesh we start by reading the .ply file, starting by finding the number of vertices and faces of this model, defined by specific lines that start with word element. Once we have those we move to the end of header and start reading the lines. First we have the definition of all vertices that the mesh is built of, then there are definitions of faces where each is defined by 3 points from the vertices list. We create a Triangle pointer for each face and store them in a vector structure that TriMesh then uses to calculate the intersection.

Since TriMesh is just many triangles, to calculate the intersection hit we simply iterate over all triangles that we have in this mesh and find the closest hit over them all. We either pick smallest positive hit distance to define where the ray intersects, or if there is no such hit we simply return a miss.

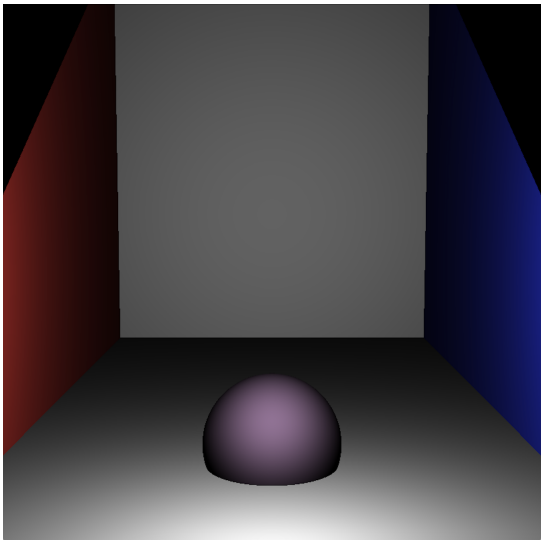

Figure 2: Dividing by distance.

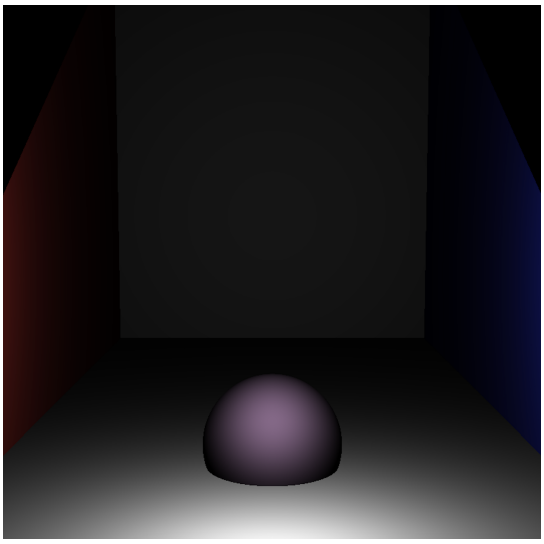

Figure 3: Dividing by distance squared.

Figure 4: Example scene objects are rendered using Blinn Phong shading.

An example render of teapot mesh can be seen in the figure 1 again where the teapot in blue is shown in the last subfigure.

TriMesh can also be used as a BVH (explained later) meaning that instead of storing and using the Triangles from the vector data structure, TriMesh uses BVH* parameter that takes care of all Triangles. When calling trimesh's intersect function and useBVH is set to true meaning the BVH is set up with this mesh, TriMesh's intersect simply redirects to BVH by calling BVH.intersect instead and returns the result of that function call instead of looping through the Triangles by itself.

All TriMesh examples shown in this report where download as .ply files from the website here.

2.4 Materials & Shading

Now that we have defined how to find the intersections of all shapes we can move to defining what happens after we find out that ray hits something. As mentioned in Shape description, each Shape object carries a Material pointer property defining how the Shape's surface looks like. Material class is used in a same way that Shape class is used, it is an abstract class which can be used by using class factory Material::createMaterial(parameters) which then returns a Material pointer to the subclass BlinnPhong as that is currently the only Material subclass that we have. BlinnPhong is defined by parameters ks, kd, kr (optional), specular exponent, diffuse color, and sometimes also texture parameters. These are then all saved as BlinnPhong class parameters and used when calculating shading using Blinn Phong shading formula.

Color of the hit point depends on several things, some are stored in the material itself, such as ks, kd, kr parameters, but some are described by the light such as id and is, and some are described by the hit and ray structures, for example hit normal and ray origin. These are all needed in the Blinn Phong formula and so instead of passing all parameters around to the material or violating the UML class diagram shown in the assignment, the calculation of Blinn Phong is split between multiple places. First we find all parameters we need from the scene's castRay function (following Blinn Phong notation here and Phong formula notation here)this are: hit normal N, viewing direction V, light direction L_m to light source m and H direction as defined by Blinn Phong shading, $H = L_m \cdot V$. Once we define these, we use them to calculate the specular and diffuse component as diffuse equal to $(N \cdot L_m) * id_m$ and specular equal to $N \cdot H$ and send these two results with the hit point, is_m that is the is parameter of the light source and the distance between hit point and light to the getMaterialColor function implemented by the shape that this hit corresponds to. This function then points to the material which takes all these parameters

and combines them with its own parameters to finally produce the color of the hit. The final diffuse component is calculated as $dif = kd * diffuse$ and specular component as $spec = ks * specular^{specexp} * is$. The final color returned is defined as $((dif + spec) * baseColor)/dist$ where baseColor is either the diffuseColor parameter that is part of the shape's json description or it is a color taken from texture (next section). We also further modify the color based on how far the hit is from the light source using the dist parameter, because we want objects further away to have smaller color values, thus be darker.

The result created by using Blinn Phong shading included the distance-to-light division of the value is shown on the figure 4. The subfigure on the left shows the result when dividing the Blinn Phong components with distance between light and the hit point, we can see that the furthest plane still seems a bit too bright so another approach is to use distance squared instead, which results in a render shown on the 2nd subfigure. In my opinion, using distance squared makes the image too dark, so the rest of the figures will be created using just distance, as I believe the next features and more visible in that way.

Once we have the color returned we add it to the color we already had calculated (0 if this is first or only light source, previous light source color if there are more) and then if the material also has kr component meaning it is reflective, we cast a new ray with Secondary ray type to find the color of reflected hit, and then add this color multiplied by the material reflectness kr and add to our previously calculated color again. Furthermore, we also cast a shadow ray towards the light to see if it is obstructed and thus we should have a hard shadow instead of the color found, this is done by repeatedly (why? later in AreaLight) sending ray from hit point to the light and keeping track of how many times it hits some other object before it hits the light.

To find the reflected ray's direction we follow the math formula of it inspired by this article and calculate it as $R = rDir - (2 * (rDir \cdot N) * N)$, where $\cdot$ means dot product and $*$ corresponds to scalar multiplication. We need to also add a little bit of noise to the origin of this new reflected ray that we are creating because sometimes due to floating point error the hit point can be slightly below the surface it hit and if that happens the ray hits the surface that it was supposed to originate from. We move the ray origin $rayO$ slightly away from the surface by adding small noise to it in the direction of the normal $hitN$ of the hit surface point: $noise = 0.0001 * hitN$, $rayO = hitP + noise$. We then cast this ray with origin defined by $rayO$ and direction defined by normalized R and add the resulting color of the castRay output multiplied by reflectness parameter kr to our so far calculated color (from previous steps of primary ray color calculation).

Now that we added reflections the resulting image using the default input file can be seen in figure 7. Since we are using only 1 bounce, we see reflection of the side triangles in the floor plane but not in the back plane because this would be reflection in the reflection meaning 2 bounces of the reflected ray.

To check for obstructions we perform similar steps as with reflection. We create new shadow ray with origin being in the hit point plus some small noise, and direction now being normalized vector from the hit point to the light source position. We cast multiple rays towards the light and check the closest t returned by the function getClosestIntersectionT(ray) for each of them, if it is larger than length of hit point to light vector length, we add 1 to our parameter shadowStrength. Function getClosestIntersectionT simply loops over all shapes and returns closest hit t where it has hit some shape or infinity it it has missed all of them. Once we cast all rays, we multiply the color of hit found so far by shadowStrength divide by number of rays we casted. This means if all of them didn't intersect any obstruction before they reached light, we simply multiply the color found so far by 1, keeping it the same. Otherwise it is shaded by that ratio.

A result of adding shadows is shown in figure 7, 2nd subfigure on the right. We can see there is a hard shadow around the sphere's edges. Also since we have reflections working, we can see that the back plane also shows reflection of the shadow that is behind the sphere, showing that both reflections and shadows are working correctly.

3 Texture Mapping

Now when we are able to get the color of hit using the Blinn Phong shading, the next step is to use it not on just with the constant color defined in input file but also on the color of the texture defined by a .ppm file if the Material has one. Here even though given textures were in jpg format, they first had to be opened in Gimp and saved as .ppm ASCII files so that the we can get the RGB values directly by reading the file line by line. The texture then

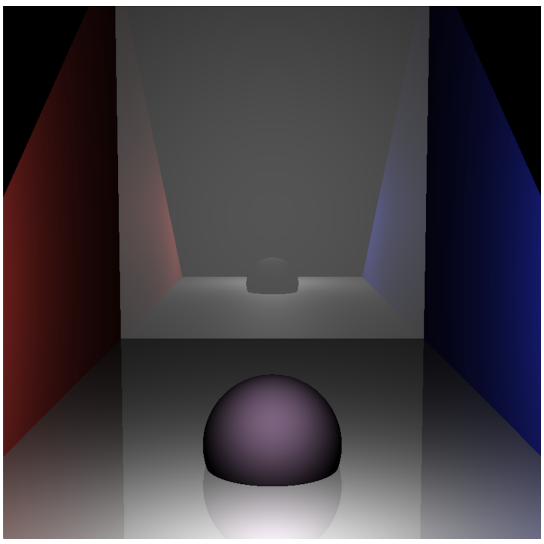

Figure 5: Adding reflections.

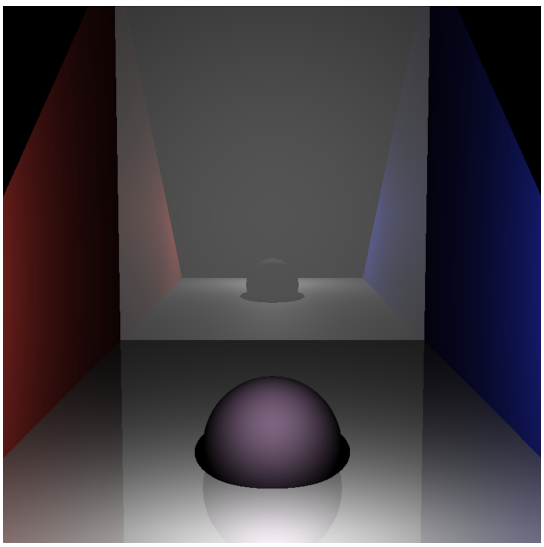

Figure 6: Adding reflections and shadows.

Figure 7: Example scene objects rendered to include reflected and shadow rays.

is defined as a vector (C++ vector) of normalized R, G and B values (math Vec3f vector, 0 to 1 range), similarly to what our resulting image is defined by before tonemapping. We store this vector of RGB values in the Material class and use it when performing texture mapping in each shape's subclass.

Texture mapping is defined by finding the u and v coordinates (range 0 to 1) that define the pixel coordinates on the texture image that we want to access and return as a base color of the hit point. Each shape has a different calculation of these coordinates but once we have them, we go to the material, multiply each uv coordinate by the width and height of the texture image to find the coordinates in the texture space. Then we find the corresponding color values from texture using $texture[u + v * width]$ because texture is a flattened vector. Texture's parameters tWidth and tHeight are read from json input file and thus must be correct, corresponding to the ppm sizes.

3.1 Sphere

Texture mapping of a sphere is inspired by the math formulas defined in Wikipedia UV mapping page. The formula used in the Sphere code is however slightly different because their coordinate system is not the same as ours. The v value is defined as $v = 0.5 * atan2(p_x, -p_y)/2 * \pi$ and u as $u = 0.5 - asin(p_z)/\pi$ where point p is a normalized vector from sphere's center to the hit point. Resulting rendered image include sphere texture mapping is shown in the first subfigure in the figure 11 using the example input provided. Note that it had to be adjusted because the specified tWidth parameter of the texture was incorrect, it was set to 850 but correct is 852. This is also clear from the .ppm file once the .jpg texture is transformed to it. With tWidth set to 850, texture will be mapped incorrectly.

3.2 Planar quad

Finding the uv coordinates on the Plane is simply done by projecting a vector from hit point to the corner to the edges of the plane $pr_u = (p - v3) \cdot (v2 - v3)/||v2 - v3||$ and then calculating the ratio of this projection and the edge length: $u = pr_u/||(v2 - v3)||$. Where the point p is the hit point on the surface we want to get the color for. For v this is the same, we just just the other edge perpendicular to the previous one: $pr_v = (p - v3) \cdot (v0 - v3)/||(v0 - v3)||$, $v = pr_v/||(v0 - v3)||$. Then we return u and $1 - v$ to match the orientation of the mapping we want to see. Resulting render is shown in the 2nd subfigure in the figure 11. We can see the texture is mapped correctly, without stretching or squeezing the picture we want to see. Moreover, the reflection in the back plane is also correct and once zoomed shows that front of the floor plane is also correctly mapped.

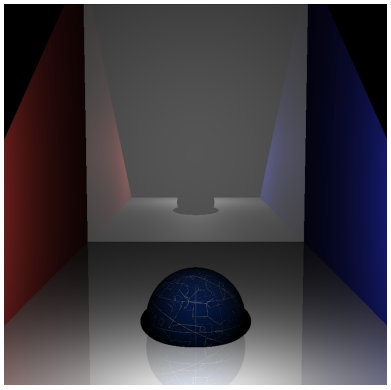

Figure 8: Sphere uv mapping.

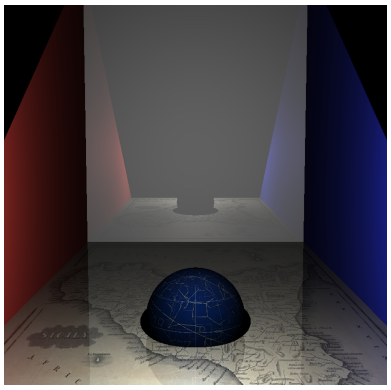

Figure 9: Plane uv mapping.

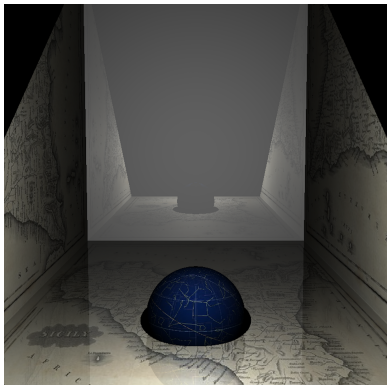

Figure 10: Triangle uv mapping.

Figure 11: Example scene objects rendered to include textures.

3.3 Triangle

To find the triangle uv coordinates we can't simply find the projections of hit point vectors because our edges are not always perpendicular and the u and v coordinates can thus easily end up in a range outside of the $0 - 1$. Instead, we need to find the bounding box of the triangle first and project the hit point on these instead, similarly to what we did in Plane uv mapping. To find the height and width of triangle we perform some math (probably over-complicated) using the triangle area calculated using the the normal of the triangle. There are many equations here that are just for finding the width and height, and then finding the width and height vectors using Pythagoras theorem and so on (that's why probably overly complicated so not going to include all equations here). In this case we return $1 - u$ and v to match the orientation of the texture to our preference. Third subfigure in figure 11 shows the result of using the floor plane's texture also on both the side triangle walls. We can see the textures are again mapped correctly, the orientation can be changed by changing the u and v parameters, for example by not returning $1 - u$ but just u directly.

3.4 Triangle mesh

Due to time constraints triangle mesh texture mapping wasn't implemented. When thinking about it, simplest approach would be to map each triangle individually which is probably not what we want. The approach of manually splitting the texture into triangles based on the mesh vertices seems complex and time consuming so was moved to later but I never had a chance to come back to it. However, I think I read somewhere that some .ply models have a reference at each triangle face to a corresponding uv coordinate (or something like that) so given such file maybe the texture mapping wouldn't be so complex, if we at least know roughly the position of faces on the texture in an advance.

4 Bounding Volume Hierarchy

4.1 Functionality overview

BVH is another subclass of the Shape class, however it is a special case because BVH is also defined by Shapes so we have a circular connection between them. BVH takes a vector of Shape pointers as a parameter in its constructor and uses it to create its only class parameter which is a Node* root - a Node pointer which is the root of the tree that BVH really is. The Node structure is a struct with following parameters: Bounds object, left and right Node pointers - children of this Node and isInner and isEmpty booleans that are self-explanatory.

The Bounds class is a class that is used to store the bounds of the 3D box that each Node of the BVH tree sits in, parameters of Bounds are just two Vec3f values - min that is the minimum point (e.g. closest bottom left corner) and max that is the maximum point (e.g. furthest top right corner) of the box. For example the Bounds

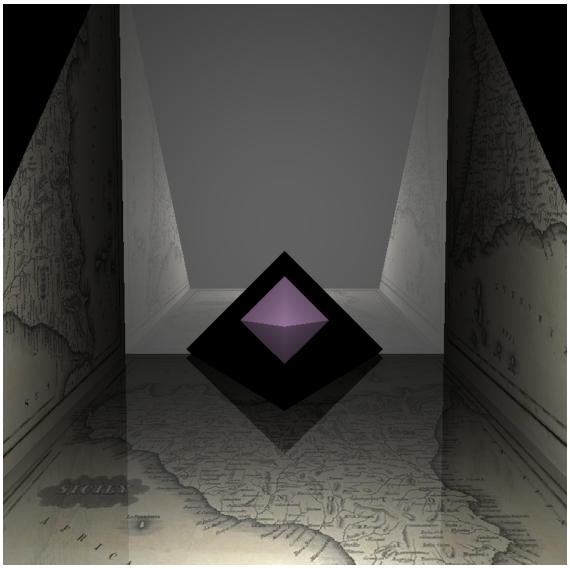

Figure 12: Octahedron TriMesh using BVH.

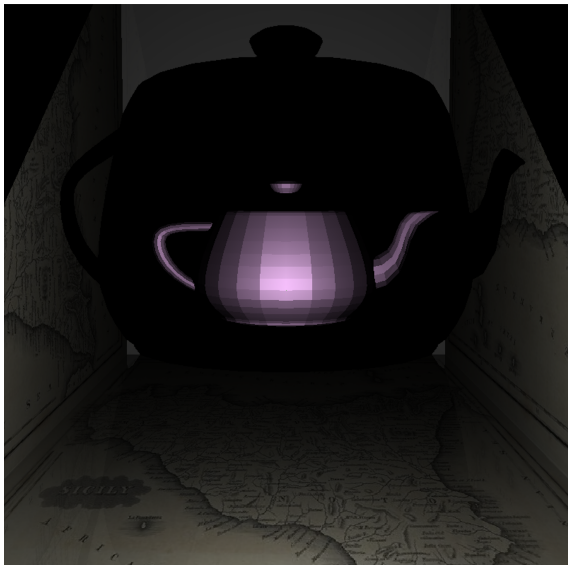

Figure 13: Teapot TriMesh using BVH.

Figure 14: Example TriMeshes rendered using BVH for both all scene's shapes and for TriMesh itself.

of the root node is a box that surrounds all shapes in the scene, finding the smallest closest point in all shapes as minimum, and furthest largest point as maximum. For this purpose, every Shape subclass implements a function called getBounds which returns the Bounds object representing bounding box of the shape. For example for Sphere, this is simply max = center + radius, min = center - radius. For Triangle and Plane we have the corner points so we just loop over them to find the largest and smallest value of each dimension and use those as max and min, respectively. For BVH this is simply the Bounds of the root and for TriMesh we loop over all vertices stored and if we find a larger/smaller value of the current bound found so far we simply update them to these newly found max/min axis values and continue (Bounds.comparePoint function).

BVH once created (either directly using new BVH(..) or using Shape::createBVH(..)) a root is created and whole tree structure is built using the vector of shape pointers that are passed to the constructor. This is done using recursive buildTree function which takes vector of shapes pointers and integer parameters called 'attempts' and returns a Node pointer to the tree (or subtree, or leaf) created by the call. Initially in constructor we pass all the shapes to the very first buildTree call, and then in recursive calls inside the buildTree we split the shapes and pass subsets of them down the tree until we can no longer split it further and store them in the leaf node, setting isInner to false and isEmpty based on if we have any shapes to store or not. Sometimes when shapes are overlapping it is not possible to store them individually and have leaf Node with just 1 shape pointer. Because of this we keep track of attempts integer indicating how many times we have tried to split this shapes vector and when we pass 1 attempt we store them all together in one leaf, creating a bounding box as a union of all of them.

The way that we split shapes into left and right child was kept simple, but should be easy to extend. We first find the bounds of all shapes that we want to split using Bounds::boundsUnion function and then uses bounds.getBoundsSplitIndex to get the axis we want to split the bounding box on. Currently, this is a simple function which returns the longest axis in the bounding box, meaning we always cut the bounding box in half along the longest side, we use the midpoint (max-min / 2) to do this. I also tried just picking index randomly but the performance didn't seem to improve. Once we have the splitting index we iterate over all shapes and compare if their centroid point (mid point, same as before the shape's bounds is max minus min divided by 2) at the splitting index and if it is smaller it will be send to left child, otherwise to the right child. And this process goes on until we either have just one shape pointer left (or 0 if all end up in another child when splitting), or are unable to split further (attempts = MAX_ATTEMPTS), that's when we return a leaf and the recursions starts rolling back from the bottom up filling all nodes until we return the root.

Once BVH is build, we use it in the same way as any other Shape subclass, using intersect function to calcu-

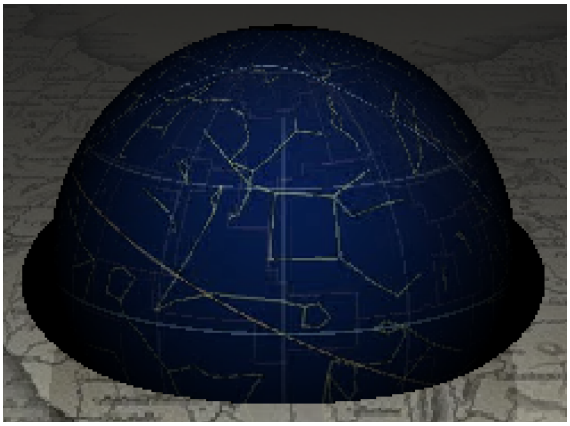

Figure 15: PointLight shadow.

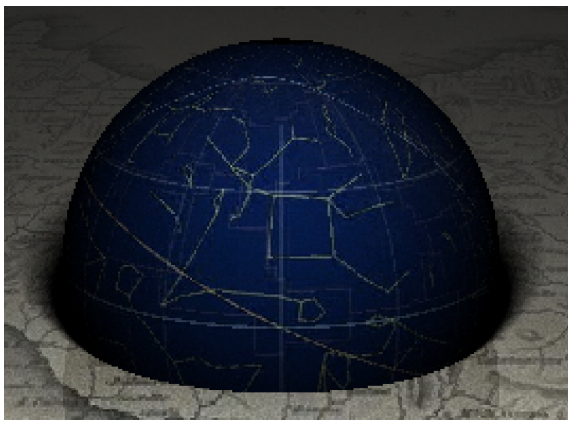

Figure 16: AreaLight shadow.

Figure 17: Comparison of shadow created when using PointLight (left) versus AreaLight(right).

late the intersection point and return the Hit structure. Function intersect(ray) implemented by the BVH calls the intersectTree(node, ray) function with node being the root and then this function is recursively called again as we go down the tree to find which leaf this ray intersects. If we are not at the leaf yet, we check if the ray intersects the bounds using bounds.intersect() for both children of this node and if that's true and ray trully intersects the child's node bounds, we call intersectTree again with this child node to get the Hit from them. Once we have the hit (or don't if there is none), we compare them (if at least one child has returned something) to see which hit is closer by analysing the hit t parameter, making sure it is also positive so we have a hit in front of the ray origin, we then return it back to the intersect of BVH which returns it back to scene where it was called from.

Bounds intersect function is implemented to perform ray-box intersection by looking at intersections of the ray with all 6 planes that surround the box. For each we get a position where ray hits the bounding box plane and what we need to know is if this hit at any dimension happens before existing any of the bounds other dimensions. This is again a bit complex and confusing, involving several values calculations (different t values for all dimensions of min and max values of bounds) and then comparing them to against each other, so I will not include details here, however the output is simple: returns false if the ray misses the box, true if it hits it. This approach is explained in this notes called AABB intersection and also in this tutorial where they also show some code (very confusing though so was adjusted by a lot).

4.2 Performance comparison

Even though I couldn't find any mistake in my BVH implementation, it is clearly the case that it is either inefficiently set up or incorrectly used because the rendering time of meshes is not that much better. For simple meshes such as octahedron, the rendering times was brought from 4 minutes 55 seconds to 2 minutes 11 seconds when only performing 1 attempt of node splitting. For larger meshes, such as a teapot with 2256 faces, this results in teapot to render for 6 minutes and 31 seconds where before making changes to trimesh to include BVH this was around 30 minutes. Resulting octahedron and teapot renders are shown in figure 14.

Potential improvements to BVH include a different, smarter way of splitting the space and shapes into children nodes, for example taking into account the volume we are splitting and position of each shape in it, or maybe spitting in such a way that we divide most number of shapes into two parts. This will not only create better trees with better splitted trees, but can also improve the speed of calculation of intersection by a lot. This would also mean that setting up the BVH structure will take longer time, but rendering should then be faster.

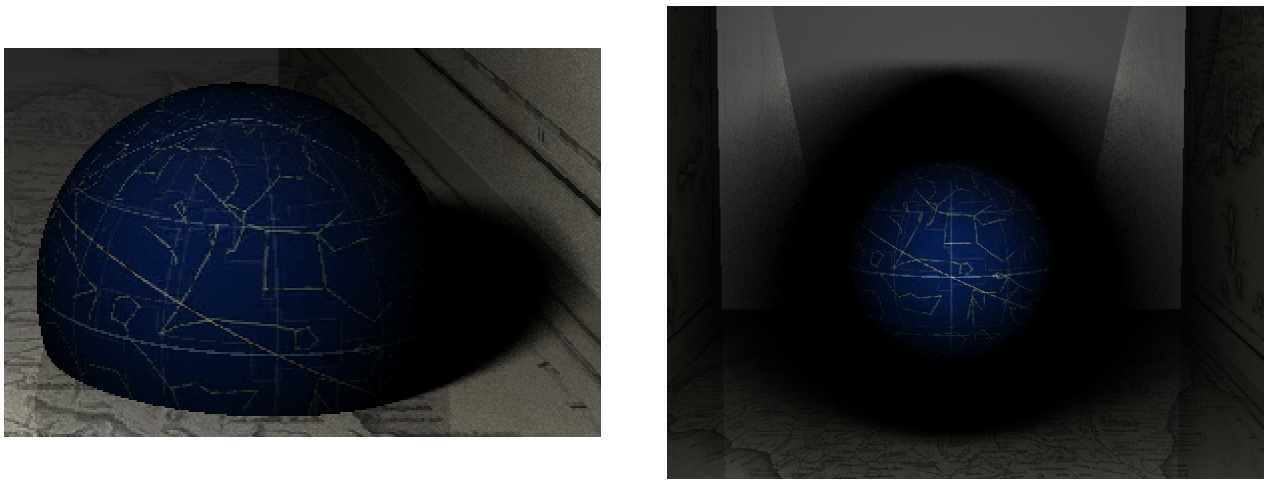

Figure 18: Comparison of AreaLight shadow created when changing the position of the sphere.

5 Distributed ray tracing

In this section due to time constraints, the only part that was attempted is implementation of the AreaLight functionalities. AreaLight is defined by 3 parameters - corners of the plane where the light is placed, meaning it has a bounded plane shape but just 3 points are needed here. AreaLight is used in such a way that each time getPosition function is called on this light, it returns a random position inside the points that define it. First we find all maximum and minimum coordinates, then based on which dimension is constant (since this is 2D object) we return a random value in the other 2 non-constant coordinates, for example if x dimension is constant (the same for all 3 points that define the AreaLight) we calculate the following: y = random(y_min,y_max) and z = random(z_min,z_max) and return the ligth position as pos = [v0.x, y, z], where v0 is one of the corners and for example y_min = min(v0.y, min(v1.y, v2.y)). The way to get random number in given range is done using rand() function and was done by the following a stackoverflow question.

Now we have light that has different position every time we ask for it, so we need to add a loop and sample the position multiple times when checking for shadows (as already introduced at the beginning of the report), then average out the result to get the soft shadows, meaning we will no longer have just black or not black, but also values in between.

The following example was done using sample size equal to 100, meaning we sample the position of light 100 times for each shadow ray calculation, resulting in slower performance, however this is done purely to illustrate the effects of the AreaLight sample so more samples create nicer comparison. The points defining the area light are as follows: "v0": [-0.2, 0.8, 1.5], "v1": [0.2, 0.8, 1.5], "v2": [0.0, 1.2, 1.5]. The effect of area light on shadows casted by the sphere in the figure 17 in right subfigure, compared to the PointLight light shadow that is in the left subfigure. We can see that the shadow is no longer hard which is what we want to see. Figure 18 shows the affect when we move sphere to the side and above the plane to see larger shadows and we can clearly see a nice shaded, non hard shadow casted on the plane. However, in the 2nd subfigure it seems that the AreaLight is slightly too big, so the next figure shows result when using smaller size of the light: "v0": [-0.1, 0.9, 1.5], "v1": [0.1, 0.9, 1.5], "v2": [0.0, 1.1, 1.5] shown on figure 19. We can see that now the shadow is still not a hard shadow, we still have regions on side where the shadow is shaded, but we have more of overlapping region where the shadow is just black.

6 Capabilities demonstration

In this section several rendered images with different shapes and parameter settings are shown. Due to time constraints and rendering time, not many were created. The final submitted render is shown on figure 20. We can see that there is some problem with meshes where few faces are black, this could be error in .ply file reading or intersection function. For some reason this didn't happen before with previous meshes so maybe I accidentally changed something in the code, but can't find any mistake now unfortunately.

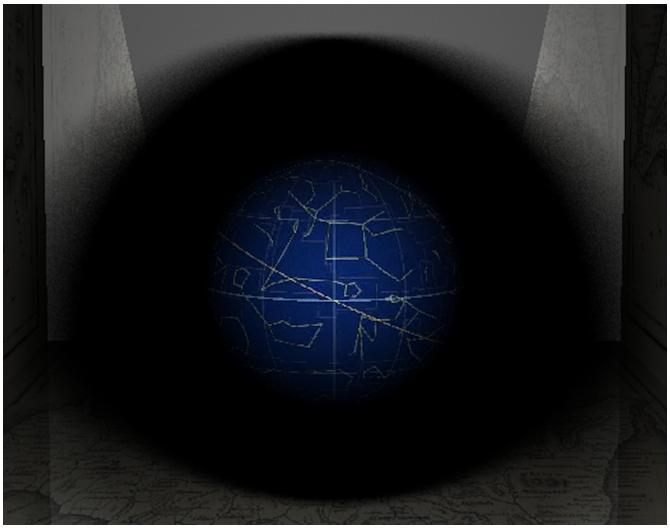

Figure 19: Shadow created when Sphere is above the plane in front of smaller AreaLight light.

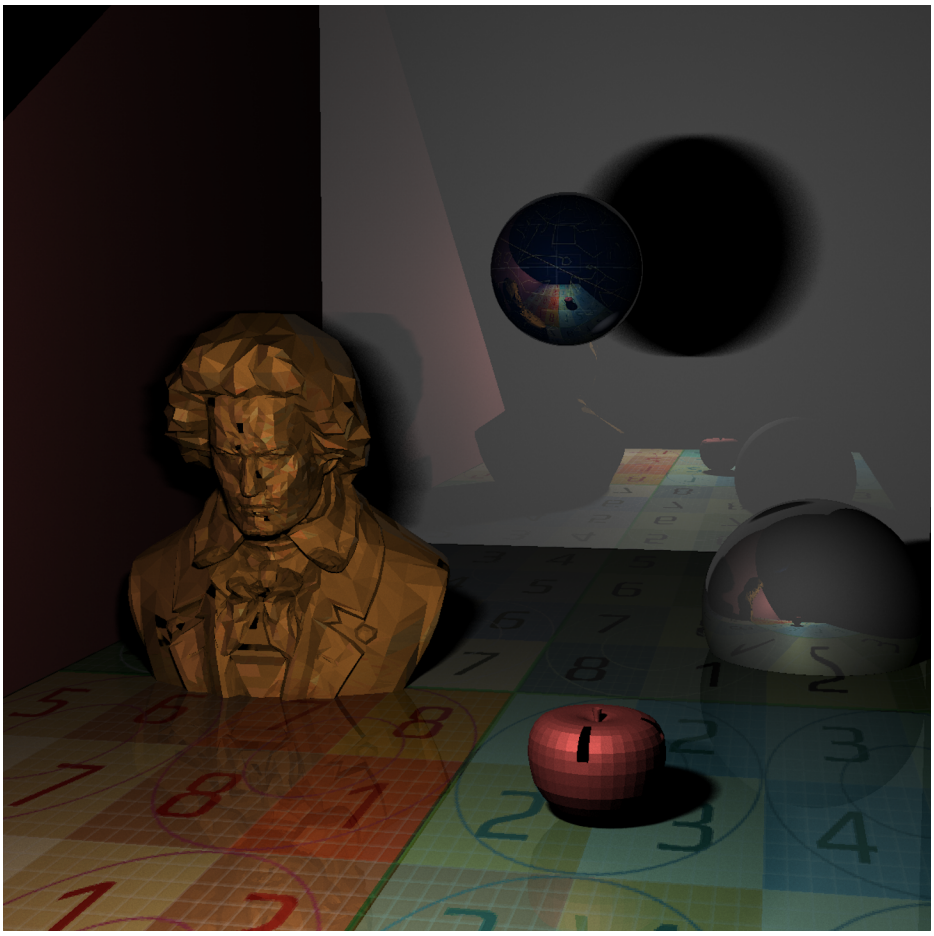

Figure 20: Final render image illustrating renderer features.

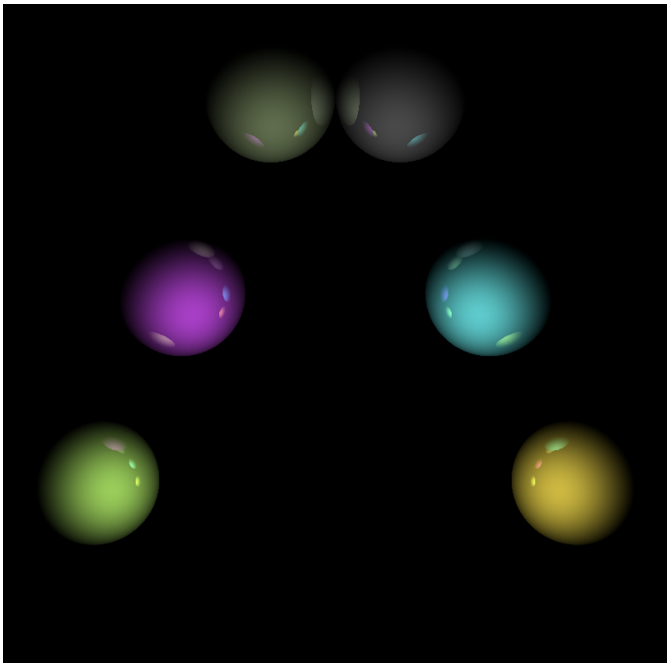

Figure 21: Scene with several spheres showing how each reflects all other spheres, with closer spheres having largest reflections, with correct orientation and color of reflections.

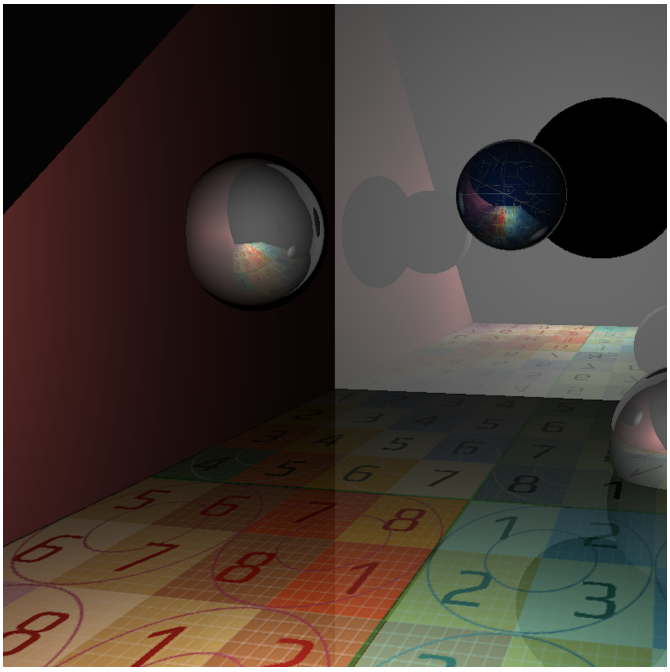

Figure 22: Scene with spheres acting as mirrors, with closer rotated camera.

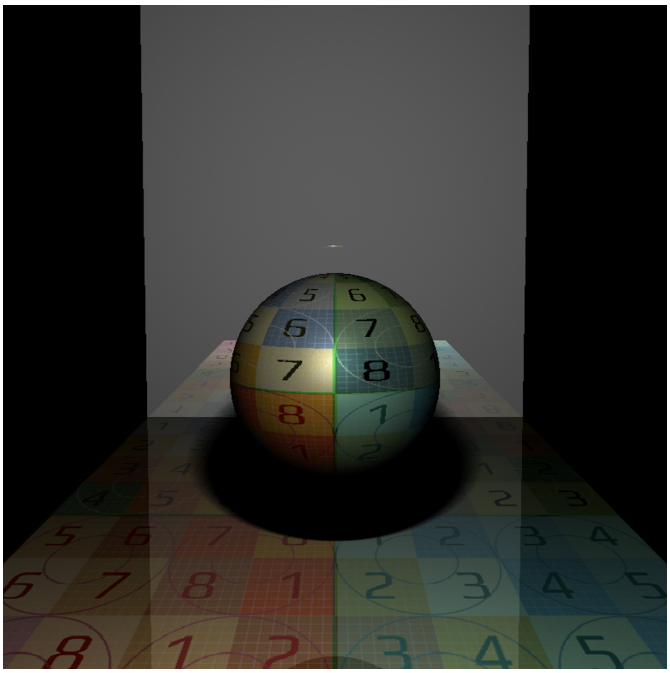

Figure 23: Scene with sphere and plane using numbered texture but with larger specular exponent and soft shadows.