Computer Graphics Assignment 2 Documentation

Youtube video for our demo: https://youtu.be/RUFY5gcM-zM

Contributions of the Team Members:

Selin Ceydeli

Minimap
Smooth Path using Bézier curves
Multiple Viewpoints
Bézier Curve Constant Speed
Multiple Shadows from Different Light Sources
Animated Textures
Particle Effects (Starry Night Stars Effect) *-Contributed equally with Lemon*

Lemon He

Particle Effects (Starry Night Stars Effect) *-Contributed equally with Selin*
PBR Shader
Material Textures
Normal mapping
Post-processing effects (glow)
Fog Effect
Particle Collision

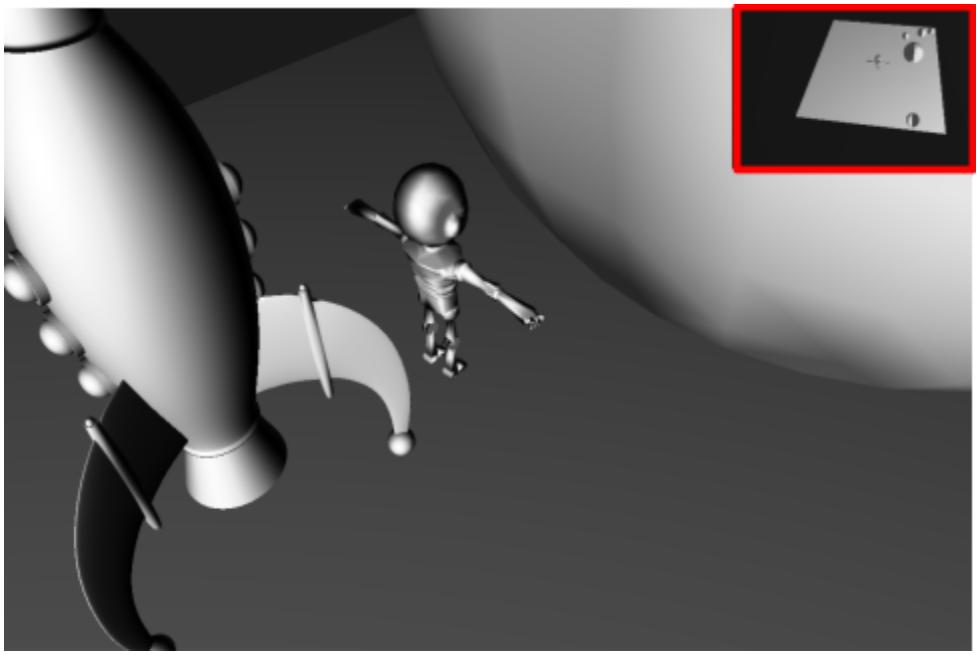
Diana Banta

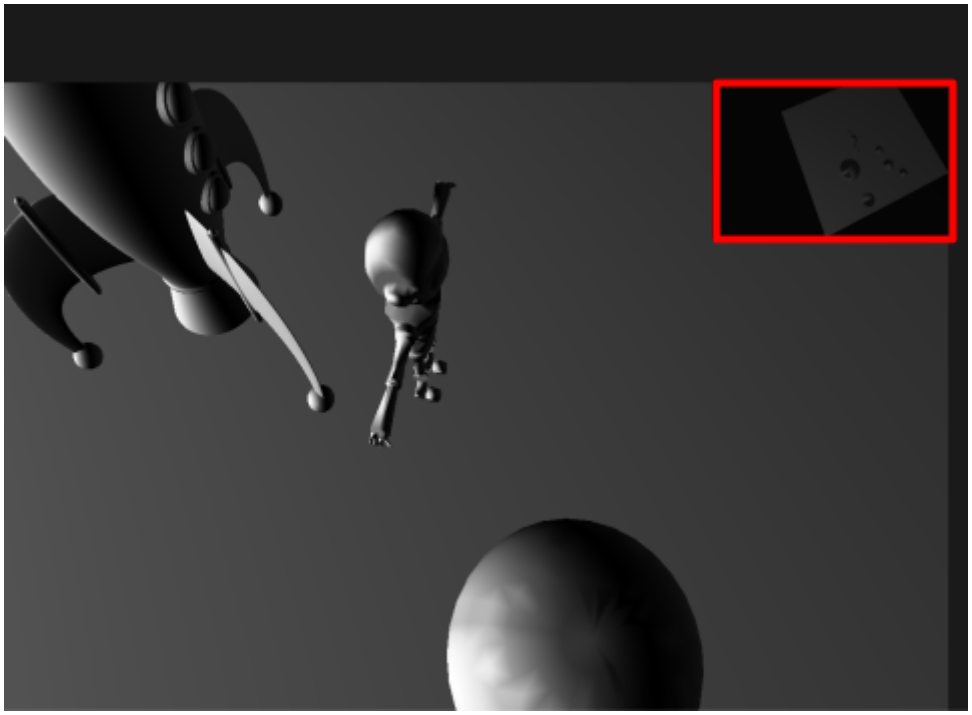
Environment Map
Hierarchical transformations
Day-Night Effect
Procedurally Generated Terrain

Minimap

The minimap is implemented in two render passes. The first render pass captures the top view of the scene. During the first rendering pass, the top view of the scene is captured under diffuse lighting with an ambient term to simulate constant, indirect light in the scene. The output of the first render pass is stored in a texture. The second render pass renders the minimap stored in the texture on a small window in the main scene (at the top-right corner).

In the pictures below, you see that the minimap is rendered on the top-right corner of the scene.



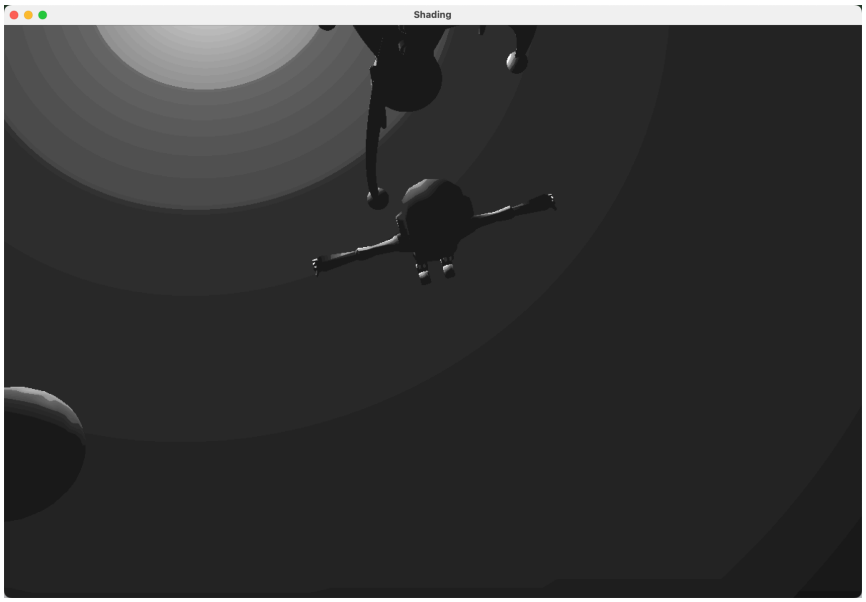


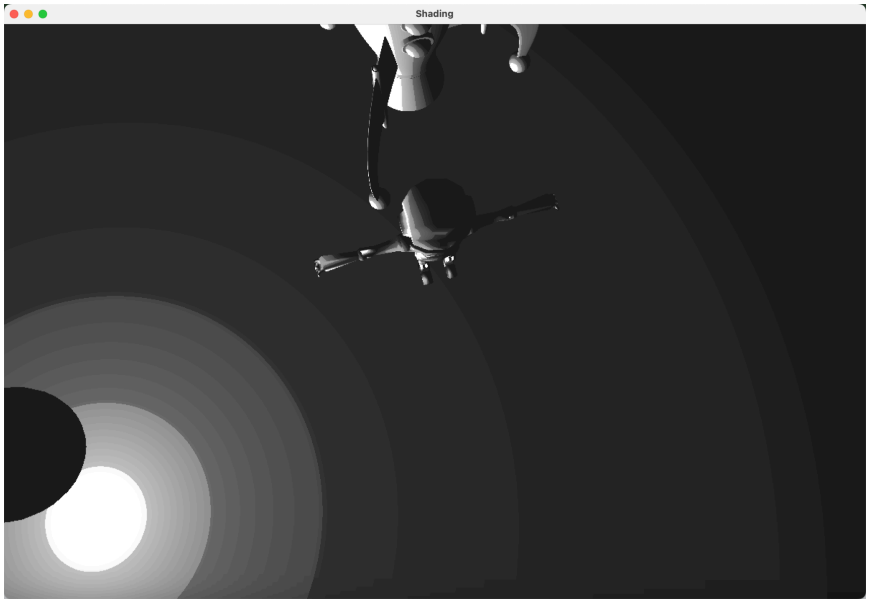
Smooth Path using Bézier curves

Smooth Path is implemented using 5 consecutive cubic Bézier curves as a trajectory for the light source. The implementation interpolates the light's position along predefined paths by calculating points based on control points. The control points of the cubic Bézier curves are defined so that the light moves smoothly from one curve to the next along the 5 consecutive cubic Bézier curves.

Similarly, movement along a cubic Bézier curve is implemented for the camera. The trajectory of the camera movement follows one cubic Bézier curve to simulate a third-person camera that comes closer to the scene (this is also implemented for the purpose of the ***multiple viewpoints*** feature).

The mathematical formulation for the Cubic Bézier curves is implemented by following the “Cubic Bézier curves” section here. Also referred to the Lecture 4 - Computer Animation slides for further information about the Bézier curves.



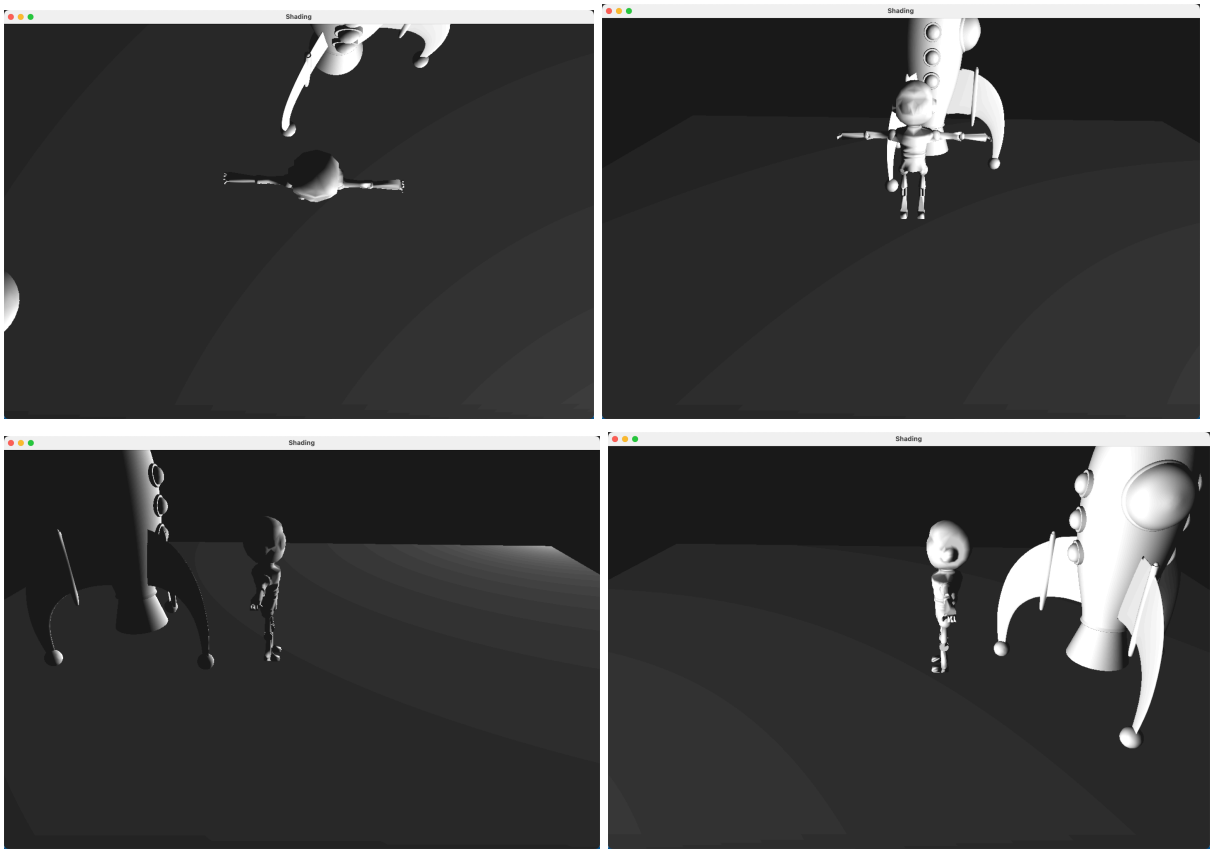


For the movement of the light along the Bézier curves, please refer to our demo video for illustration.

Multiple Viewpoints

The multiple viewpoints feature is implemented by defining separate Trackball camera instances for different perspectives: main, top, right, left, and front views. Each camera is configured with specific rotation angles and distances to dynamically provide different views of the scene. To achieve dynamic switching of cameras, a pointer for each camera is defined, and the active camera instance is dynamically switched to the correct camera pointer within the main rendering loop.

From left to right:
Top View, Front View
Left View, Right View

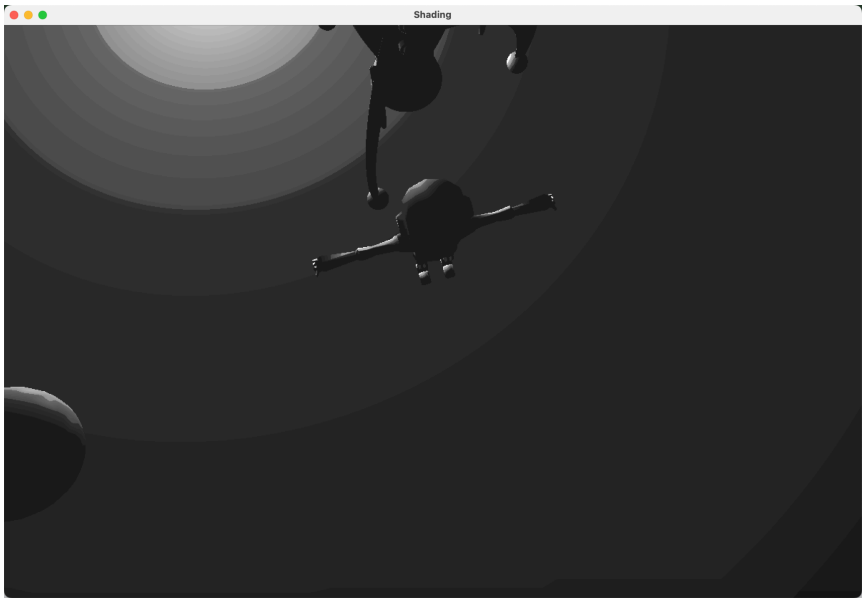


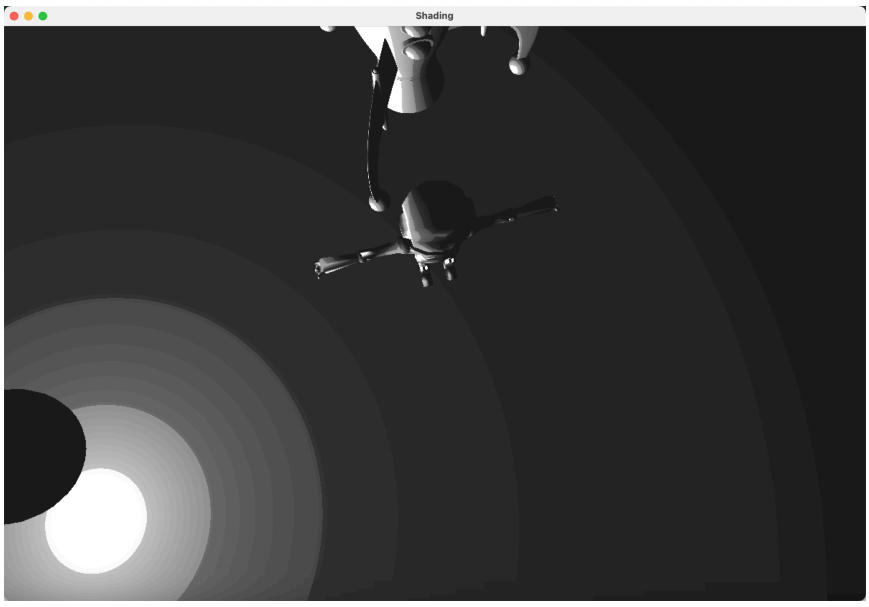
Bézier Curve Constant Speed

A constant speed motion along the Bézier curve is implemented for light movement in the following steps. Firstly, evenly spaced points along each curve segment is calculated. These points are then stored in an array for creating a precomputed path for smooth traversal. Lastly, the position of the light source is updated along the recomputed path, interpolating between consecutive points based on the elapsed time and maintaining a consistent speed. Interpolation between consecutive points was implemented to smooth the path because without it, the linear segments in the light movement were visible, making the motion appear unrealistic.

To conceptually understand the constant speed motion along the Bézier curve, this article is read.

For the movement of the light along the Bézier curves with a constant speed, please refer to our demo video for illustration.



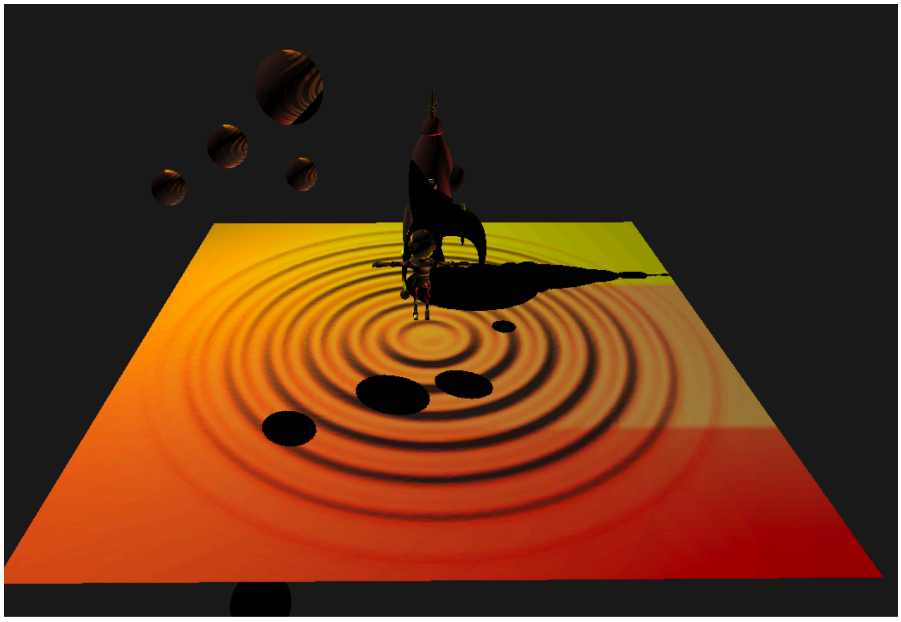


Multiple Shadows from Different Light Sources

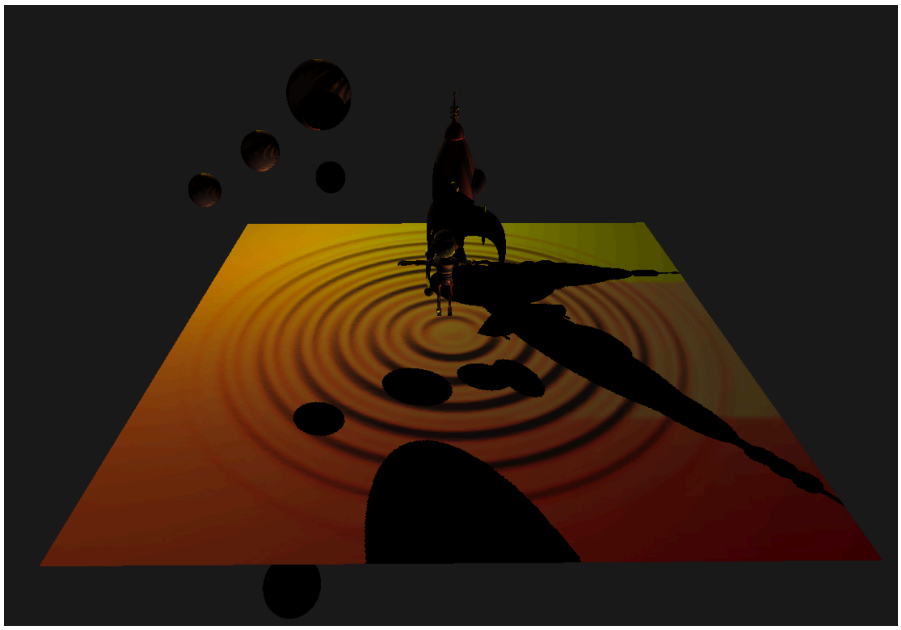
To achieve multiple shadows from different light sources, a secondary light source is added to the scene. A secondary shadow texture is then created, along with a new framebuffer for this texture. The shadow maps for both the primary and secondary light sources are rendered sequentially.

Once both shadow maps are generated, they are combined in the final rendering pass. This is done by sampling from both shadow textures in the fragment shader and applying shadow calculations for each light source independently. The resulting shadow contributions are then blended to create the effect of multiple shadows interacting on the scene.

Before adding multiple shadows (shadows are generated only using the primary light source):

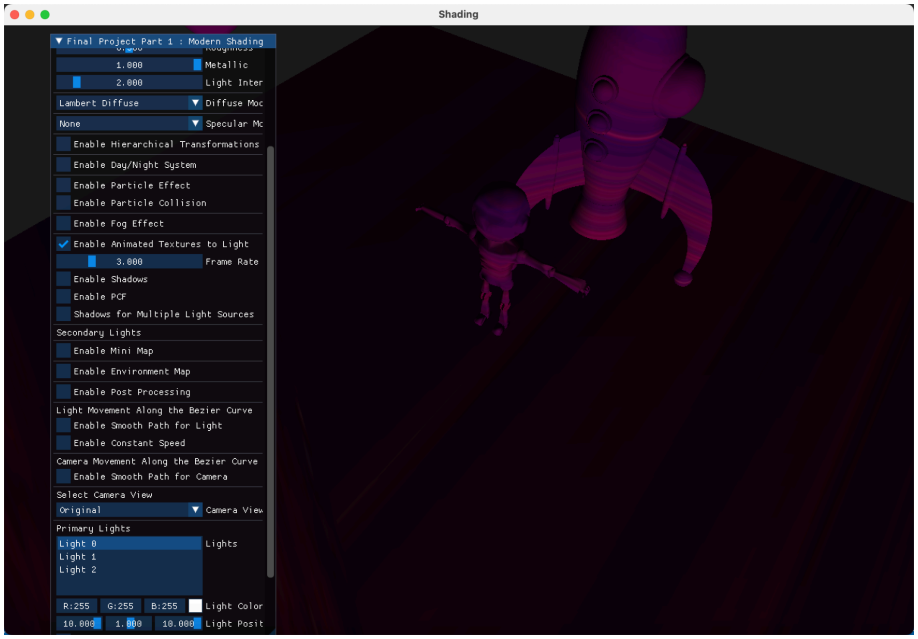


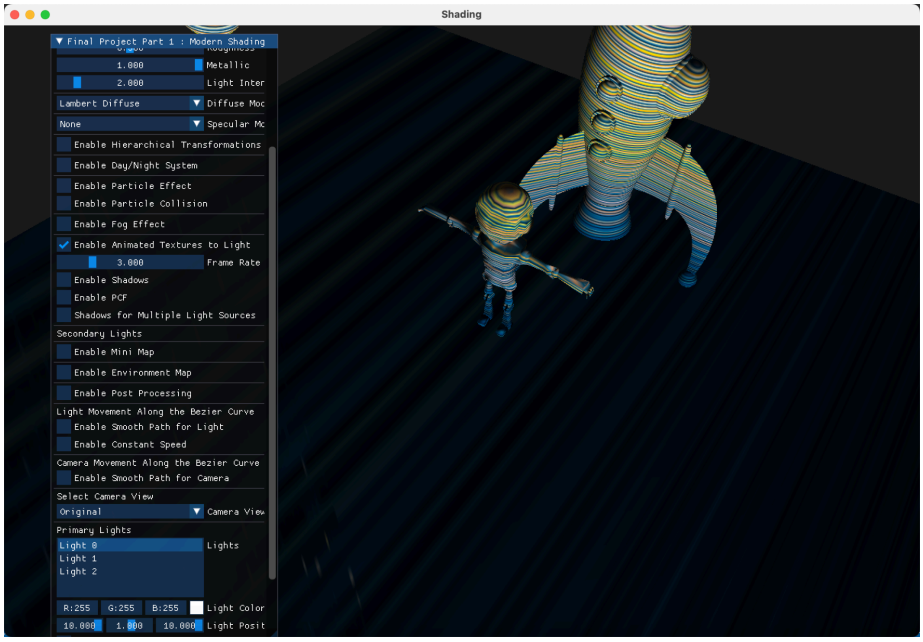
After adding multiple shadows using the secondary light source, the scene looks like below:



Animated Textures

To simulate animated textures, the textures are stored in a vector. On each frame update, the time elapsed since the last texture change is calculated. If this elapsed time exceeds the specified frame interval, the next texture in the sequence is displayed, and the light color is updated accordingly. This approach ensures consistent frame updates and allows users to control the animation speed through the UI. Additionally, users can adjust the frame rate per second, ensuring that the animation runs smoothly at an appropriate framerate, independent of the computer's performance.



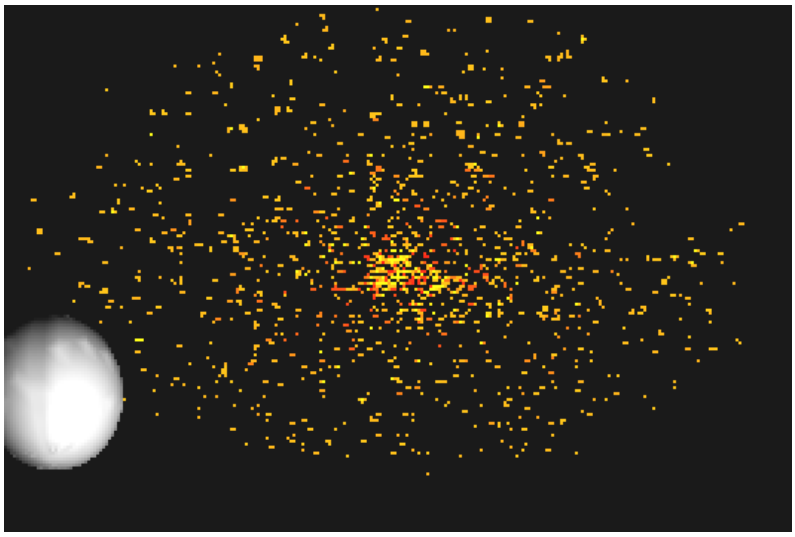


Particle Effects (Starry Night Stars Effect)

To create a starry effect above our objects, we emit particles that stay on a fixed plane above them. Once generated, the particles will be given initial life span, speed, color and position. Upon every frame, we will check if some particles have walked past their entire life, they will be replaced by new-born particles from the emitter. During each update, the particles' properties gradually change to create a dynamic, shimmering effect. This continuous cycle of birth, transformation, and replacement simulates starry particle effects that add a magical ambiance to the scene.

Particle effect is seen on the top of the scene, simulating red and yellow stars.

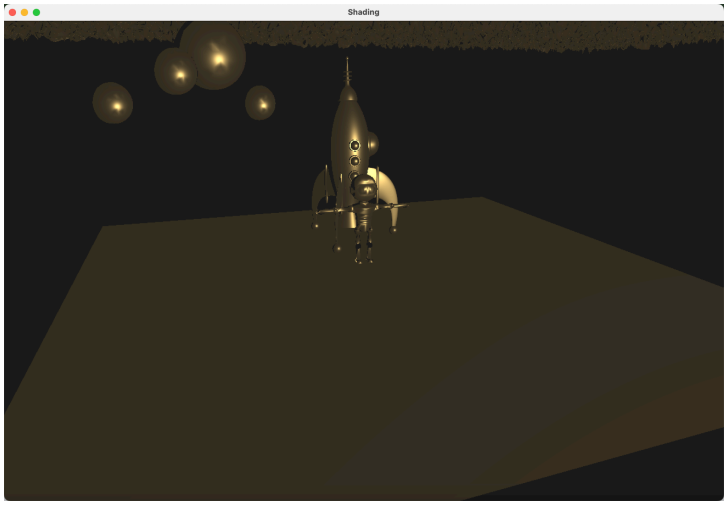
This is the close view of the particle effects:



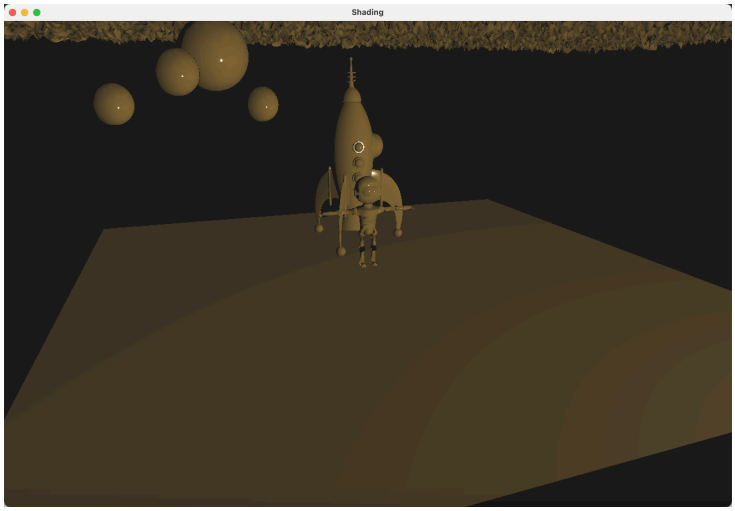
PBR Shader

The PBR shader computes lights into three parts, separately specular light, diffuse light and ambient light.
For the theory part, I referred to the Real Shading in Unreal Engine 4.
For the calculation, the shader begins by calculating the view, light, and halfway vectors, which are essential for shading calculations based on the fragment's position and normal.
The specular reflection is computed using the Cook-Torrance model, which combines the Fresnel effect, normal distribution function and geometry function to simulate realistic reflections. The Fresnel effect adjusts reflectivity based on the view angle, the NDF models the microfacet distribution across the surface, and the geometry function accounts for the shadowing and masking of light between microfacets.
Diffuse lighting is implemented as a simple Lambertian reflection, modulated by the material's albedo color.
The ambient term is a simple constant, representing indirect light to ensure areas without direct illumination aren't fully dark.
Finally, we sum the three components up to achieve a physically realistic output.

The first one shows a more metallic output.



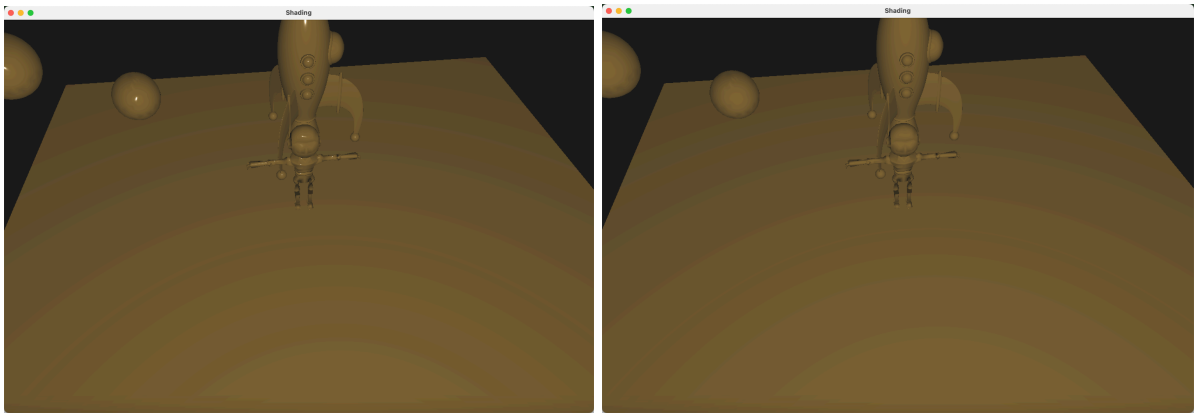
While the second one shows a more plastic like output.



Material Textures

The new Materials for the textures we have currently include albedo(the base color), roughness(controls the sharpness of specular highlights), metallic(defines the material's reflectivity), and light intensity(parameter to scale the radiance).
By using these parameters in the PBR shader, we can easily create metal-like textures or normal plastic ones.

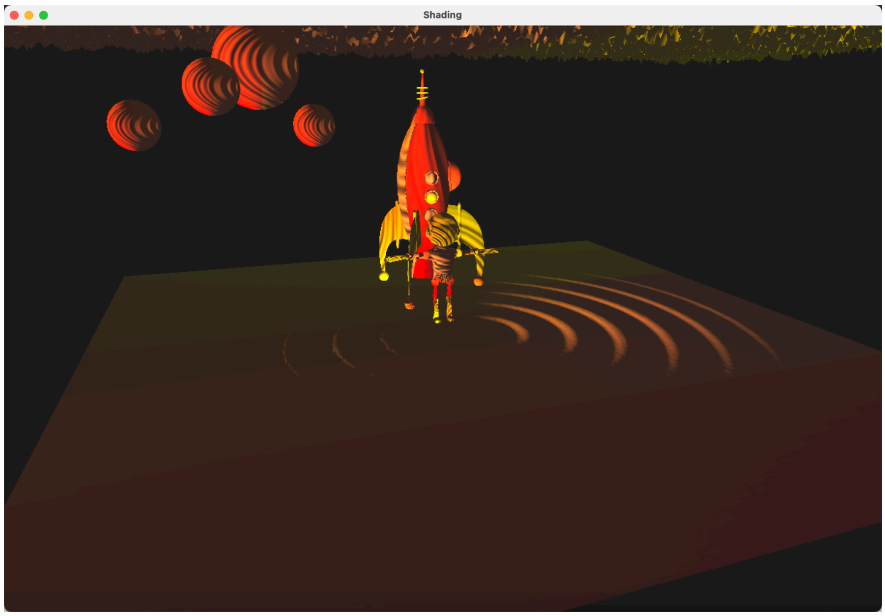
From left to right: roughtness = 0.2, roughness = 0.9.

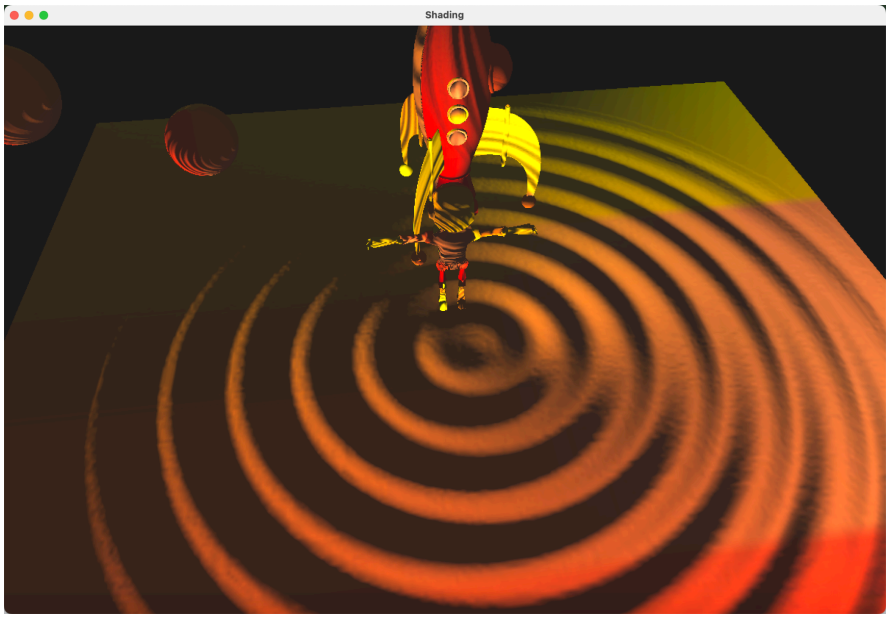


Normal mapping

Instead of using the normal value from vao, we use the normal map, which has the value of the normals on each point stored on the x, y, and z channels in the image. Since the normals are in the tangent space, we need to transform them into world space. To achieve this, we calculate tangent and bitangent vectors for each vertex to form a Tangent-Bitangent-Normal (TBN) matrix. This matrix transforms the normals from tangent space to world space, allowing the normal map's details to be applied correctly in the scene's lighting calculations. Finally, we sample the normal map for each fragment and transform it using the TBN matrix, integrating it seamlessly into the lighting model for added surface detail. Since the normals are pre-defined, we can achieve a three-dimensional effect on a plane with little extra calculation.

Two pictures demonstrates different mapping with different light positions.

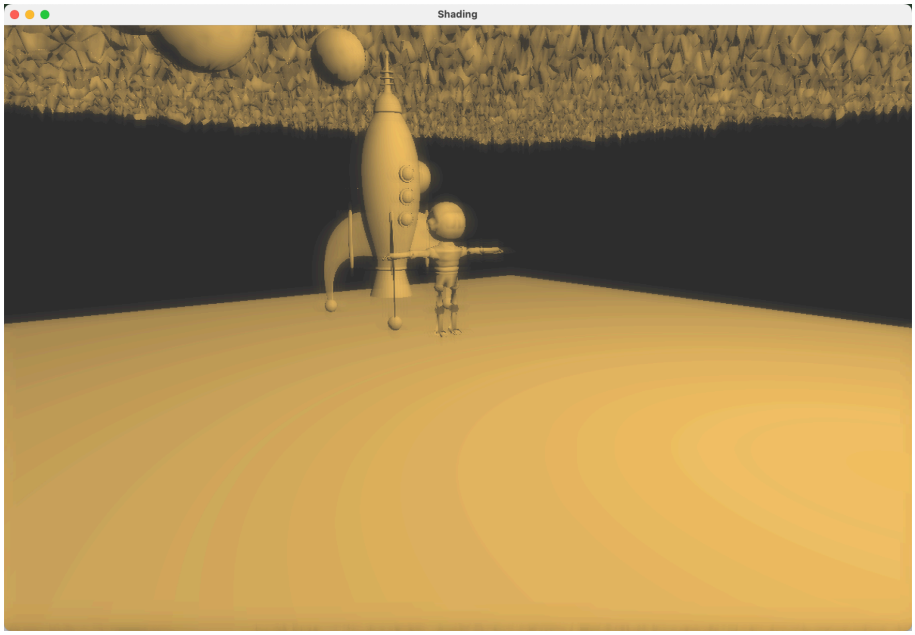


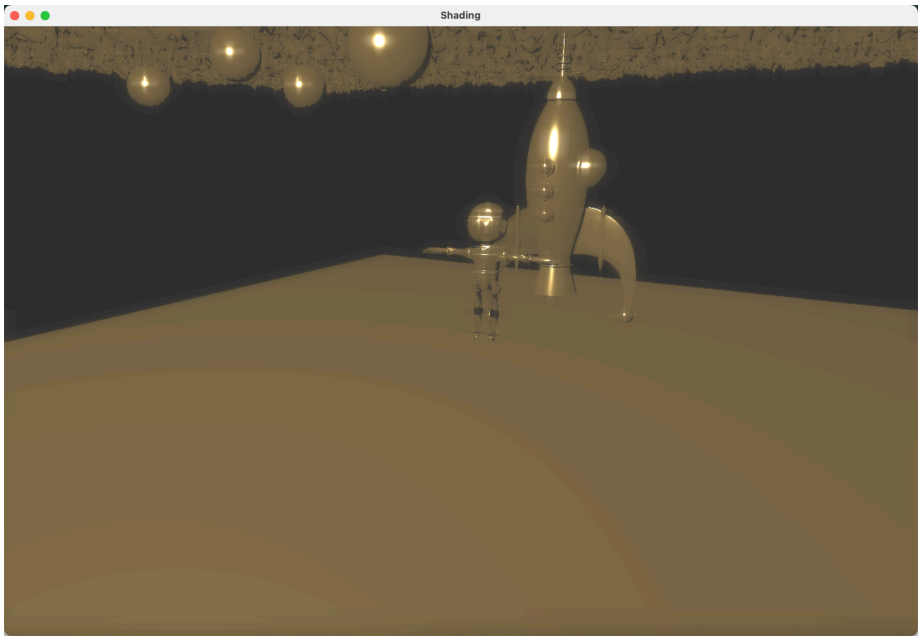


Post-processing effects (glow)

We're using a double render pass to do the post-processing. So first we create a frame buffer and bind a screen texture to store the current result of the first render pass. Then we use this texture in our second render pass as the base color. In the second render, we isolate the brightest parts of a scene, applying a blur to create a soft halo, and then adding this halo back onto the original image. Through this, we can create a soft light around bright areas, making objects look like they emit light.

The two images shows glow effect with different metallic value.

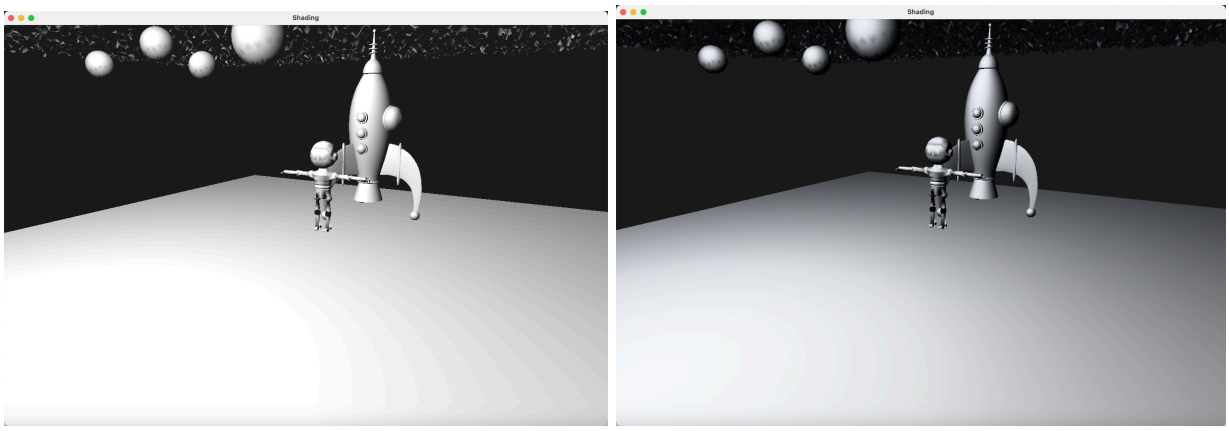




Fog Effect

For the fog effect, we define a threshold to control the fog's spatial range and use an exponential function to calculate the fog factor. Meanwhile, we give a periodical value of the fog density sampled over time to add movement, creating the illusion of fog drifting through the scene and then fading away.

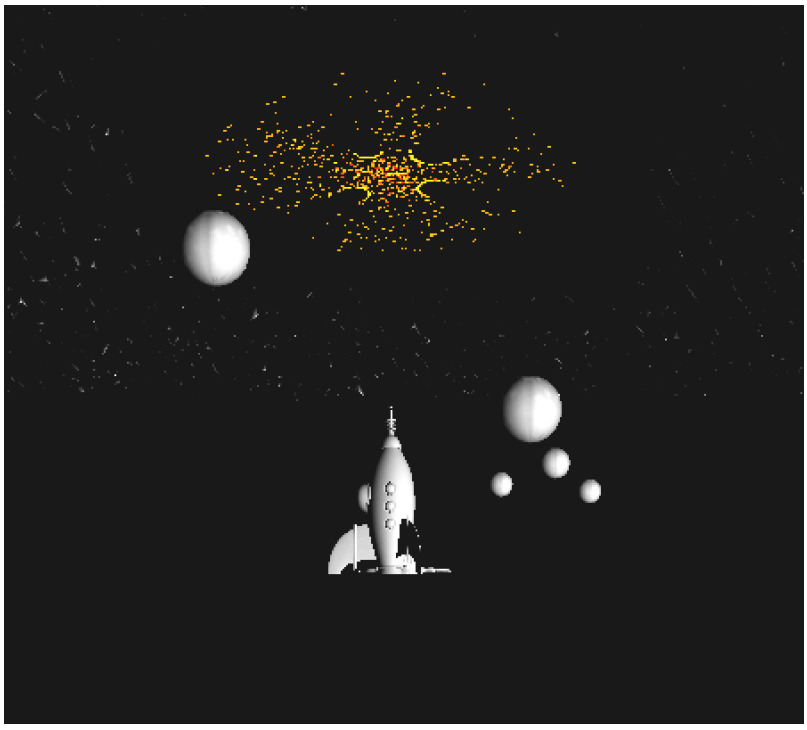
Left: Without fog, Right: With fog

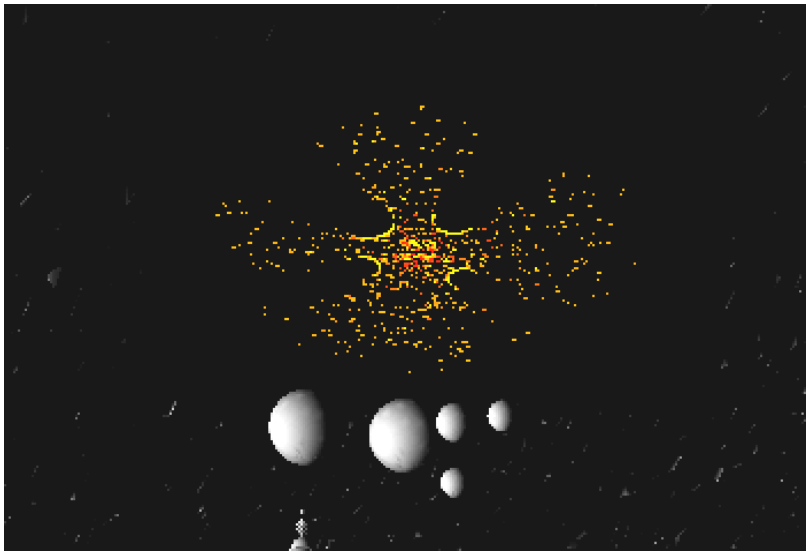


Particle Collision

To enhance the realism of our particle system, we added collision detection with obstacles.
In this scene, we added 4 circle obstacles around the emitter. For each particle, we calculate the distance between the particle and the obstacles on every update. If this distance is less than or equal to the obstacle's radius, a collision is detected, then the particle's velocity is adjusted to simulate a bounce or slide along the obstacle's surface. Also, to simulate energy loss upon impact, we scale down the particle's velocity slightly, adding realism to the collision response.

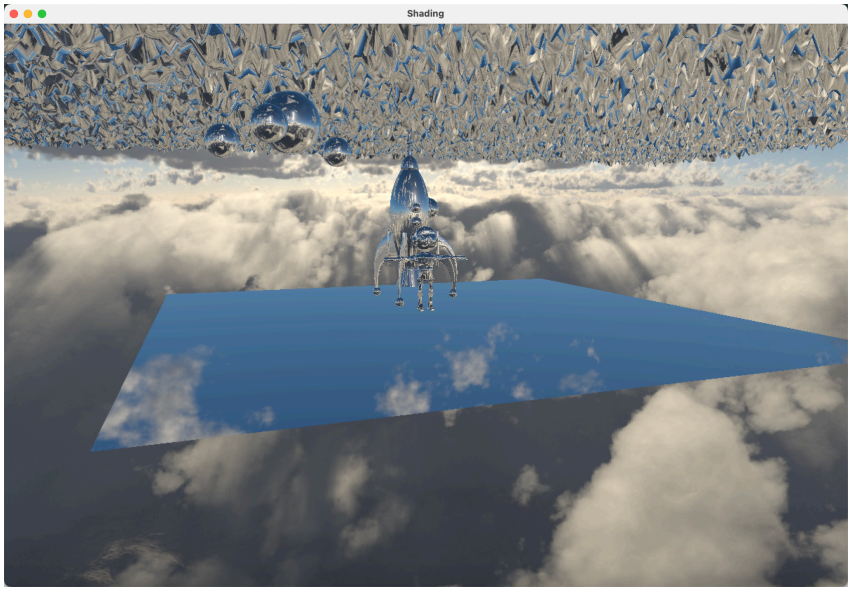
Once colliding the 4 circle obstacles, the particles will be rebounced.

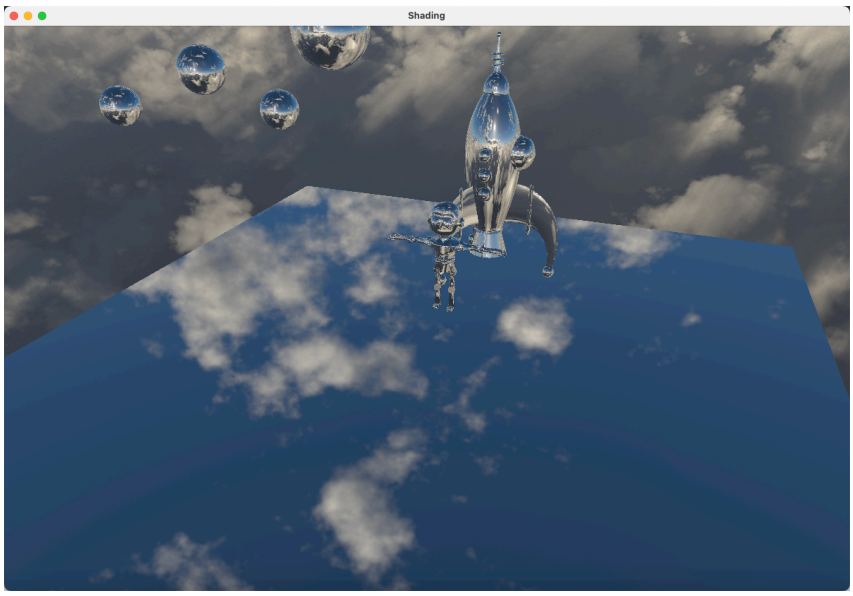




Environment Map

The environment map is done by first creating a cubemap texture from 6 square 2D texture images. This is done by iterating through each image and then binding it to one face of the cube. Then, in the rendering loop, the cubemap is sampled by calculating the reflection angle from the position of the object to the camera, and the corresponding color value is looked up from the cubemap texture. The final result is meshes that have a reflective aspect.

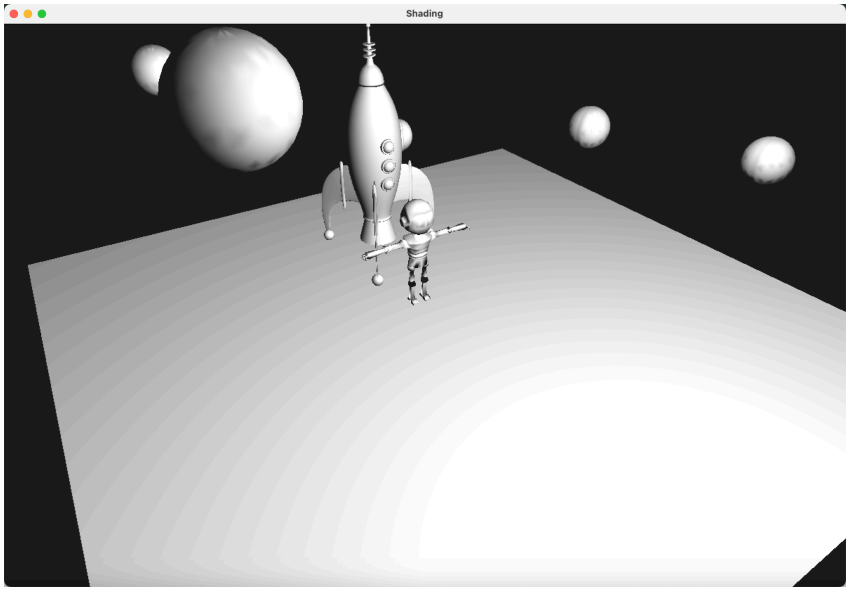


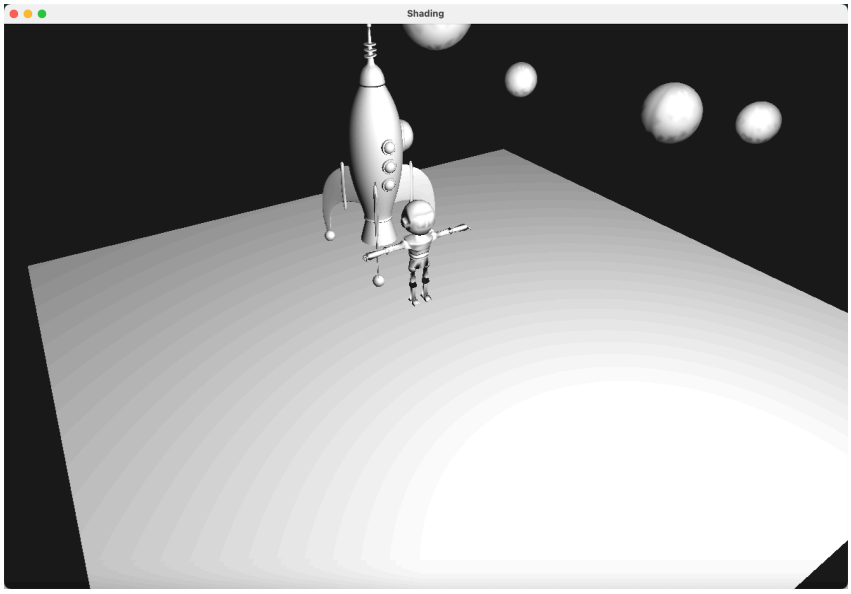


Hierarchical transformations

For better demonstration, please refer to the video.

The hierarchical transformations feature works by adding 4 more meshes, representing the planets. “Big sphere” influences all of the others, while “Medium sphere” influences the two “Small meshes”. For each iteration of the render loop, a “time” parameter is calculated, and this is used to control the transformations and rotations of the spheres. Every time, the positions of the spheres are recomputed and updated in the scene. The child meshes use the new computed position of the parents in the calculation, to ensure that the dependency relationship is preserved. Once the new positions are calculated, for each vertex in each mesh the position is updated. Lastly, the buffers for the scene are updated to draw the new results.

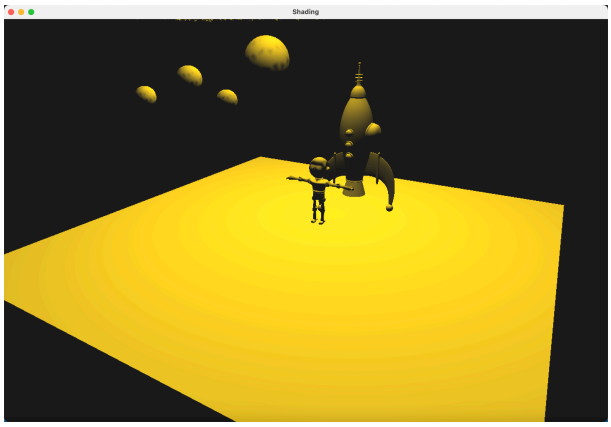


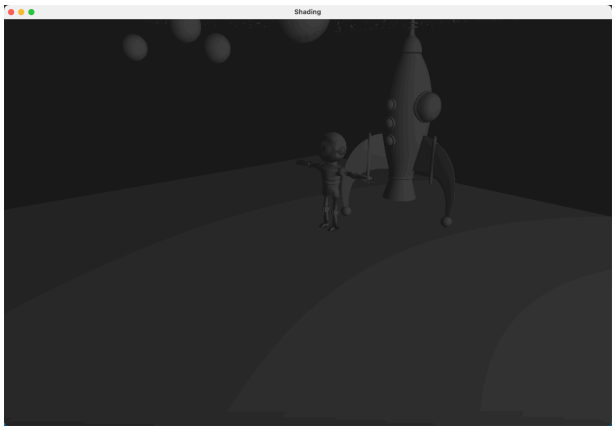


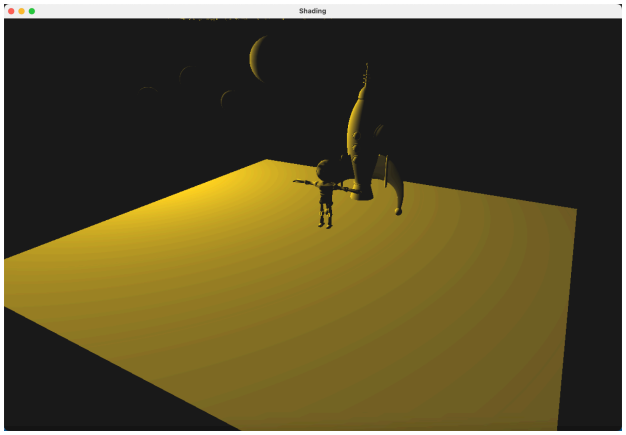
Day-Night Effect

This feature is implemented by creating 2 meshes, and then applying the same rotation values for each. Then, for each loop of the rendering, the positions of the meshes are updated. To simulate the light changing effect, the position of one of the meshes with respect to the horizontal plane is calculated each time. Once the sign of the y-coordinate changes for the mesh, the day-night switch happens. If it is day, then the light is added/moved to the current position of the sun mesh. The same thing happens if it is night, but for the moon mesh.

The images show the day-look, night-look and the transferring from day to night.







Procedurally Generated Terrain

For the procedural terrain generation, a new mesh was created algorithmically and then added to the rest of the scene meshes. This was done by defining a grid of points and vertices, where each point on the grid is a position in space. Then, for every point on the grid, a mathematical noise function was used to generate a random corresponding height. The noise function makes use of the coordinates of the point, and then generates a pseudo-random value by generating a random gradient for each corner, and then interpolating those values. Then, from these coordinates, vertices are created and then added to triangles.