G01-Technical Report

Filipe Varela 92459 | Miguel Dauphinet Barros 102184 | Nuno Estalagem 102245

1 Graphic modelling

Our first step towards our project's goal was to model the surface, Rover, Random Rollers, and squared collider objects for which we used the *basic_geometry* code provided to us.

The surface consists in a scaled cube (with equal z and x scales but the smallest height value possible for it to be displayed). On the other hand, the Rover is the composed by two rectangles (one for the "Rover's body" and another one just to make the rover more like a car) and four scaled, translated, and rotated cylinders (the center of these rotations is the Rover's body center). This approach allows the "Rover's wheels" to rotate alongside the Rover's body and not around themselves. The Random Rollers and squared colliders are Spheres and Cube's respectively. We gave shininess, emissive, ambient, and diffuse properties to the referred objects for them to be displayed by the OpenGL. The renderScene() function is responsible for rendering each array's objects. More specifically, our renderScene() function, after setting up the environment lights and textures, calls the mainRenderScene() function, which is responsible for pushing the *MODEL* matrix and adjusting each object's scale, position and rotation angle. This later function is also responsible for rendering all the remaining objects such as the spheres with environmental cube reflection, "the bumped squared arch" or the billboards, which were also modeled by the *basic_geometry* library.

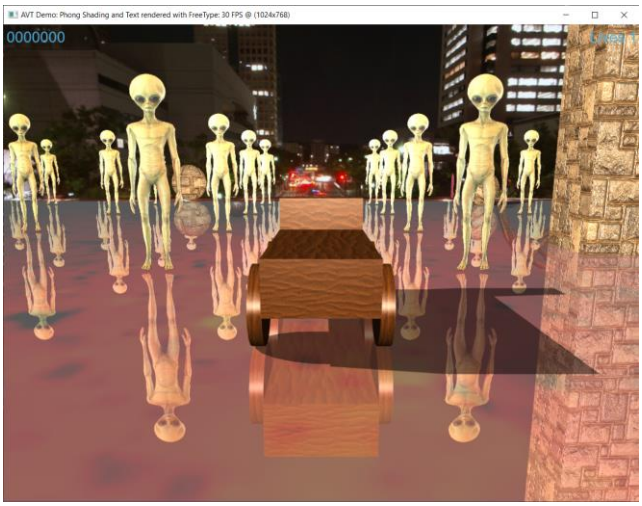

Figure 1: Overall view of the environment from the 3rd person perspective camera

2 Cameras

We developed three cameras for this project, two of which were perspective ones, and the remaining one orthogonal, where each camera captures a different part of the project environment. For our cameras, to work properly we transformed each object's World coordinates into Clipping Coordinates by using the *VIEW* Transformation Matrix. The changeSize() function is also useful to define each camera's viewport and projection.

Our setupCamera() function defines the projection each camera uses according to the selected pointer (if a user clicks the "3" key it will switch to the third person perspective camera). This function also makes use of the provided lookAt() function to define each camera's *VIEW* matrix.

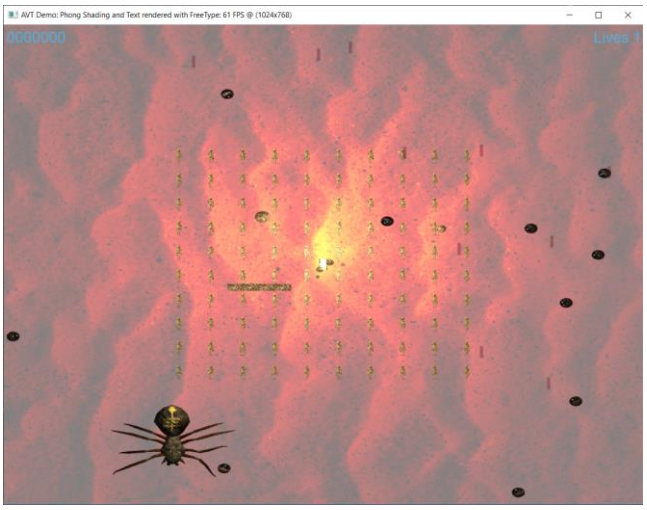

Figure 2: Orthogonal camera

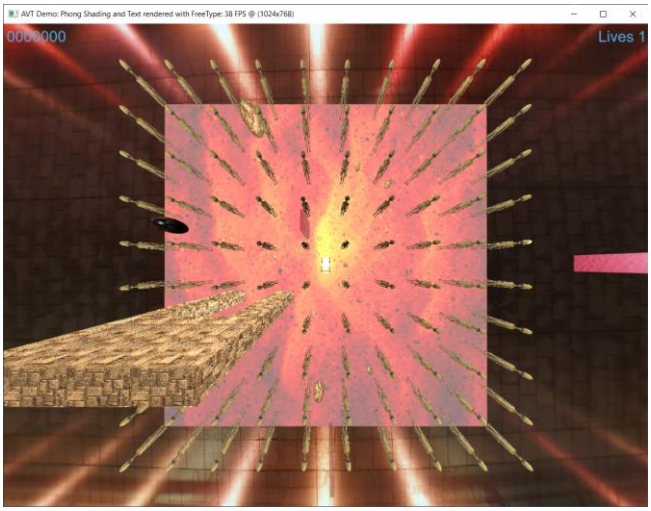

Figure 3: Non-3rd person perspective camera

3 Game Elements' movement

We developed a *process keys* function, a *listener*, which sends the received input to the respective *actuator*. When pressing the "p" key, the z-axis velocity decreases and is multiplied by a Δt, resulting in a negative angle according to right-hand rule with the y-axis, which make the rover rotate. Pressing "o" will have the opposite result ("positive angle rotation"). When "q" is pressed the x-axis velocity increases and, in case the rover angle is not zero, so does the z-axis changes accordingly with the rover's direction. The "a" button, however, decrements the x-axis velocity, with the same manner to z-axis velocity as well, which allows to back the rover up. The rover x position changes according to the velX * cosf(roverAngle * M_PI / 180.0f) * dt formula, whereas the z position changes according to the velX * -sinf(roverAngle * M_PI / 180.0f) * dt formula. The surface is flat, meaning that the rover's Y value never changes. As for the Random Rollers' velocity, this is incremented each scene is rendered up to a limit as well as its angle. The velocity and acceleration levels in general, however, are controlled by the fixedRenderScene() function, which defines the mentioned velocities according to the current game state.

4 Lighting of the scene

Firstly, we based our pointlights' implementation in the provided demos. Nonetheless, since we aimed at creating more than one pointlight, our fragment shaders' "dataIn" and our vertex shader's "dataOut" variables were both converted to an array, in order for them to support multiple lights. We are aware, though, that our project only displays four point lights. This occurs because the largest output size supported by the vertex shader's array is 8, meaning that we would not be able to display all the implemented lights.

For our headlights, we used the cutoff angle and the half vector for the lighting cone angle.

For our fragment shader, it accumulates the values of all the existing (and active) lights in the environment and compares them with the directional light (similar to the point lights, except the fact that its alpha value is 0) to calculate each fragment's coloration(colorOut).

5 Collision detection

There are two possible ways in which collisions may occur. On the one hand, our rover may be hit by a random roller, which involves the rover to return to its base position. On the other hand, the rover may also hit on of the many walls scattered across the environment, resulting in it considerably reducing its velocity and acceleration, and the wall moving towards the opposite direction it was striked at. Since our rover's body is a scaled cube and that all 4 of his wheels were placed below that same cube, we calculated each of the cube's extremities. A collision was detected whenever any part of a wall/random roller was within the interval of these limits, including the limits themselves.

6 Texture mapping

We start of by using the glGenTexturesfunction() to declare the number of textures we want to load. We then load all the used textures to 2 arrays, one for the lens flare and another one for the remaining object's textures. We do so by using the provided *Texture2D_Loader* API. Both these steps are done in the init() function. To allow multitexturing, on the other hand, we use glActiveTexture() to activate

every texture and then glBindTexture(). The later one binds a texture with the respective texture array index. The glUniform1i() function is then called to assign a texture to its respective tex_loc. Every single tex_loc will then be sent to the fragment shader. All this is done inside the setupTextures() method which is called by our renderScene() function. renderScene() also calls the setupTextures() function which sends the mentioned tex_locs to the fragment shader. To define which texture each object receives, we also pass the texMode attribute to the shaders, which will apply different textures according to the value in this attribute.

7 Transparency

We applied the transparency property to our walls by using the glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA) to blend the colors stored in the color buffer via the alpha channel. We enable the depth buffer and draw opaque objects first. Then we set the depth buffer to the “read only mode” and then we make use of blending to draw the non-opaque with objects. This approach helps prevents issues with the Depth buffer, allowing for all objects to be drawn.

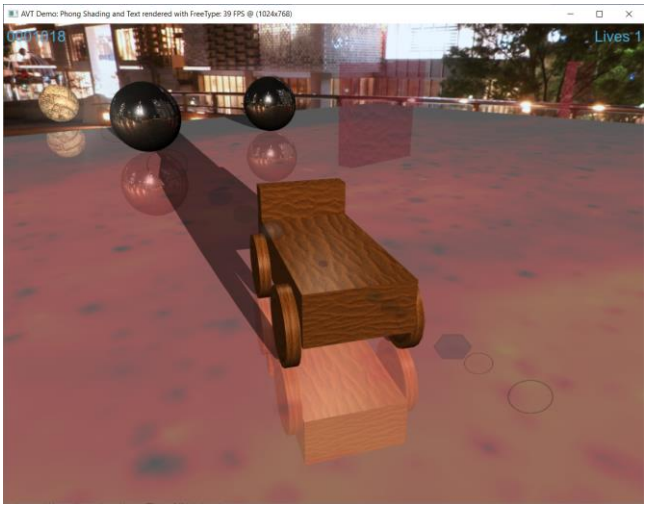

Figure 4: Textured wall with the transparency property

8 Fog Effect

To produce the fog effect, our vertex shader sends the fragments in eye coordinates to the fragment shader. The fragment shader then calculates the distance (d = length(pos)) and the visibility (vis = exp(-d * density)). The colorOut value is then calculated by mixing the visibility with the remaining colors from the colorOut and the color given to the fog (mix(vec4(fogColor, 1.0), colorOut, vis)).

9 HUD, Pause and Game Over

We based our HUD, Pause and GameOver implementation on the AVT_TEMPLATE provided to us. Therefore, our renderScene() function, after rendering all of our environment’s objects, renders the text by making use of the provided renderText() function. The renderText() function will render more or less text values according to the environment’s game state. The HUD is the “always present” text element, displaying the user’s points (which are incremented if the user manages to protect the rover and prevent it from being hit) and the number of lives the user still has. If the “s” key is pressed when the game is at the “PLAY” gamestate, our fixedRenderScene() function saves each object’s velocity and acceleration attributes and makes all objects stop (including the fireworks). Our renderScene() will also display a “Pause” text in the viewport’s middle in this scenario. Clicking the “s” key again will result in not rendering the “Pause” text as well as fixedRenderScene() “giving each object their velocity back“. The “Game Over” text is displayed in a similar fashion when compared to the pause text and informs the user that he may restart the game by pressing the “r” key. The points displayed on the HUD go back to zero whenever a life is lost. The text is rendered with an orthographic projection and with the depth test disabled to ensure that it is displayed in front of every other rendered object. The blend object is also enabled for the text to be properly rendered.

Figure 5: Fog, HUD and Pause

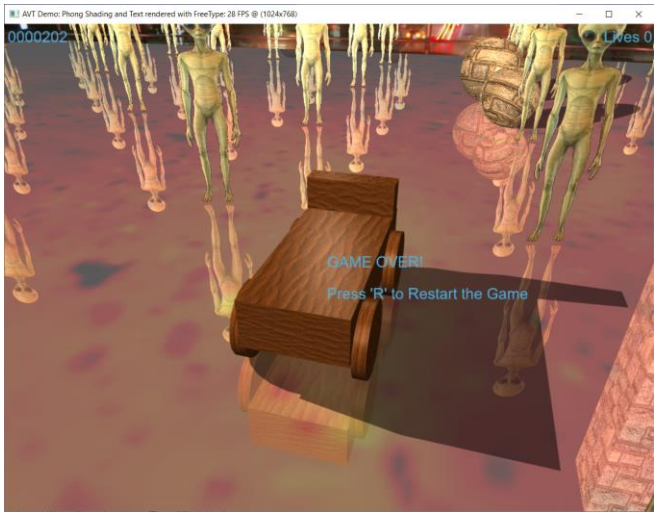

Figure 6: Game Over menu

10 OBJ objects support

For OBJ objects to be support by our project, we made use of the provided *meshfromAssimp* API, as well as the aiRecursiveRender() function. To import the Spyder OBJ we used the Import3DFromFile() function in the init() function. To render the spider we call the aiRecursiveRenderFunction() inside the mainRenderScene() function. For some unknown reason, all the textures were properly applied to the spider except the one for its head, which was outputted without any texture or color. Surprisingly, importing the remaining texture with the Texture2DLoader() function in an unused TextureArray position, solved this issue.

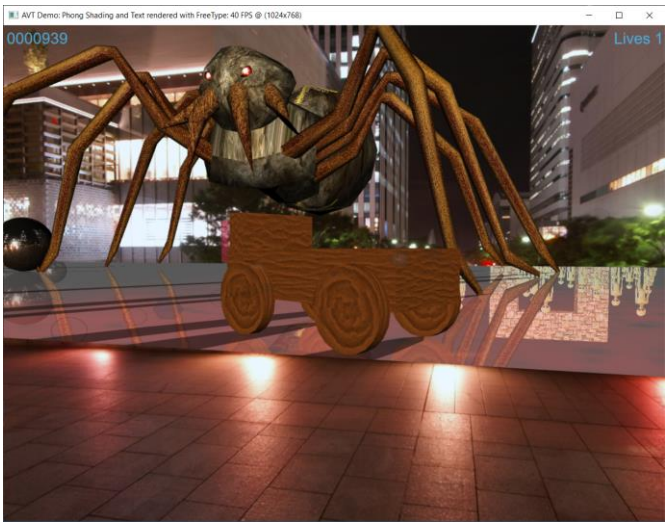

Figure 7: Spider OBJ

11 Rear-View Mirror

The stencil-buffer is essential for the rear view, which is currently the fourth active camera on the scene, accessed by pressing the character “4”. Therefore, we use glClearStencil(0x0) and glEnable(GL_STENCIL_TEST) on our init() function and glClear(GL_STENCIL_BUFFER_BIT) on our renderScene() function. The latter also calls the glStencilFunc(GL_NEVER, 0x1, 0x1) and glStencilOp(GL_REPLACE, GL_KEEP, GL_KEEP) functions to define the masked values and make the stencil test fail so that all stencil bits are replaced by 1. Afterwards, we render a cube by calling the renderRearCamera() to represent the rear view which is saved in a separate mesh. Then, the objects behind the car are drawn which is possible by inverting the camera and then rendering all the objects in the renderScene() function. The next step is reverting the camera back to the position it was before the inversion and rendering all the objects in front of the car.
The glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP), glStencilFunc(GL_EQUAL, 0x1, 0x1) and glStencilFunc(GL_NOTEQUAL, 0x1, 0x1) functions are useful as they certify that all mirrored objects are only drawn inside the mentioned cube, whereas the remaining objects are properly rendered by renderScene().

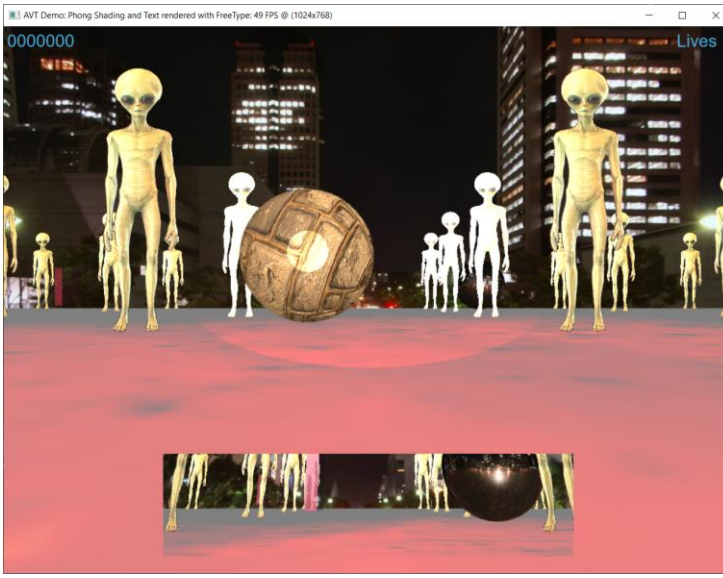

Figure 8: Rear View Camera

12 Billboard Behavior

The base for our billboards is a mere Quad to which textures were applied. To make our billboards align according to each camera’s viewing direction, we apply a specific billboard type to each camera, i.e, when a user switches from camera one to camera two, for instance, our processKeys() function sends that change to the actuators whilst also changing the billboard type. For the billboard’s background to be removed, we make use of alpha blending, discarding fragments with a 0-alpha channel. We make use of billboard type 0, 2 and 1 for cameras 1, 2 and 3 respectively, meaning we alternated between spherical and cylindrical billboarding.

Taking inspiration from the provided Billboard tutorial we made use of the Cheat Matrix reset billboard technique. The billboards are treated similarly to the rover when it comes to shaders, since their properties are influenced by the existing lights. (attempt at images 1, 2 ,3 ,5 and 6)

13 Particle System

The particles are only rendered whenever a user reaches a multiplier of 2000 score. These quad based particles, the particles' position color, acceleration, velocity and lifetime are defined in the iniParticles() function. The fireworks are rendered in the renderScene() function and their position is updated by the UpdateParticle() function. Their rendering is done with both enabled blending and a disabled depth test for the background to be blended with the quad's transparent pixels. Pressing the "s" key to pause the game will stop the particles from updating and decreasing their lifetime. On the other hand, if particles are rendered when a game ends, they are immediately discarded.

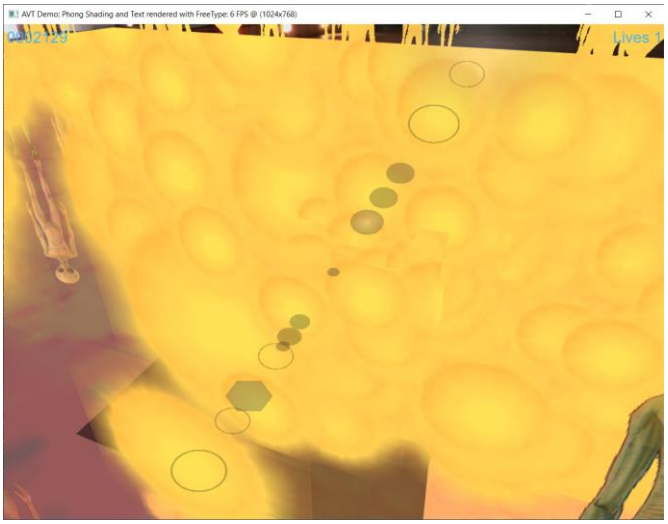

Figure 9: Particles being displayed when reaching 2000 points

14 2D Lens Flare

Our lens flare's origin is one of our existing point lights (its origin could not be the directional light, whose alpha channel is 0). The FlareTextureArray created in *lighDemo.cpp* stores the flare's textures. We based our flare's texture color in a function provided by the multiflareDemo template. Our loadFlare() function is then responsible for mapping each flare texture from the assigned light to its opposite side. The lens flare is also based in a simple quad created in the init() function, being stored in a separate array from the remaining objects. The created quad is where the flare's textures will be mapped to. Blending is enabled and the Depth test is disabled for the lens to be displayed in front of the objects.

15 Planar Shadows and Planar Reflections

For the planar shadows and reflections, we used the function reflectionsAndShadow(), isolated in the third active camera, so this coupled effect could be managed independently. First, the reflection is made by using the stencil test, filling the stencil buffer by the visible floor's geometry, and by blending a rotate version of the mainRenderScene() function and the third point light – the same one used on the lens flare – with the floor and its material data. In this last step, it was used the blend GL_SRC_ALPHA operation on the reflected image and the GL_MINUS_ONE_SRC_ALPHA operation on the floor so they could output a semi-reflective surface below the rendered objects.

As for the shadows, that is handled by the shadows() function. In it, a process in a similar manner to the reflection will occur, which in this case will be re-using the rendered objects and applying them to the shadow matrix, a transformation matrix that will convert the objects into geometry like their shadow. Then, it disables the Depth test, so it could render the shadows into the ground, and it makes use of the blend and stencil operations in order to darken the area covered by the shadow. After all these steps, the shadows are forward generated by the shadow matrix multiplied by the model matrix, in which is popped and pushed for this purpose. (attempt pictures 1,4,5,6 and 9)

NOTE: We ended up not applying shadows to our billboards, as the output was not aesthetically pleasing.

16 Bump Mapping

We applied bump mapping to both the Random Rollers and a cube composed arch. This last one was created solely to prove that our project's bump mapping was working properly, in case that wasn't noticeable enough on the Rollers and its base cube is stored in a separate array together with the "environmental cube spheres". We make use of the *stone.tga* and *normal.tga* textures, which we apply to both objects. Our vertex shader is responsible for changing the lightDir and eyeDir properties

according to tangent space. These properties will then be used by our fragment shader, resulting in a bump mapped object.

Figure 10: Bump Mapped Random Rollers and Arch

17 Skybox

The basis behind a Skybox is a cube. Therefore, we created a separate mesh to store a cube (we do so to be sure that objIds are not mixed between objects and because we can instantly assume that the objId is 0 instead of appending the cube to another array and having the trouble of finding out where the cube is placed). Each cube face will have a different texture bound to it. Therefore, our init() function makes use of the TextureCubeMap_Loader() function to load an array of 6 textures ({ "posx.jpg", "negx.jpg", "posy.jpg", "negy.jpg", "posz.jpg", "negz.jpg" }), which is then activated, bound to the Texture array and assigned to a tex_loc. Which is sent afterwards to the shaders. The vertex shader negates the skybox's x coordinates for its texture to be mapped inside the cube. The fragment shader then maps the skybox to its textures by using a cubemap variable, the one sent according to the previously mentioned tex_loc.

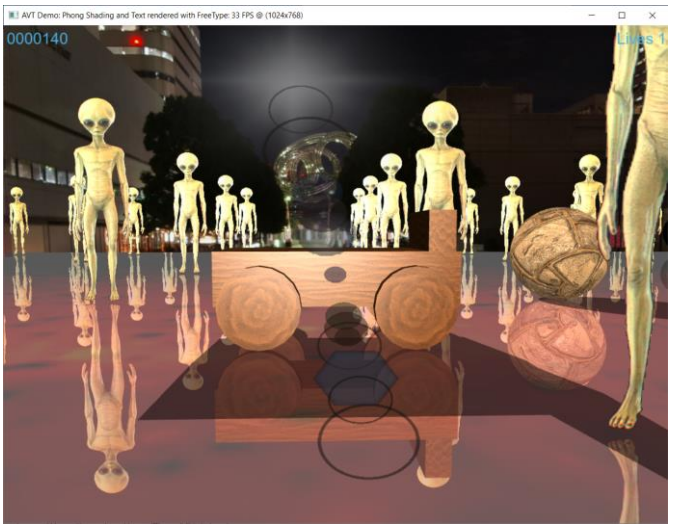

Figure 11: Skybox and Lens Flare

18 Environment Cube Mapping

As mentioned, we created spheres to which the environmental cube mapping was applied. Once again, they were stored in a separate array to simplify the objId definition and to avoid objId collisions. The texture applied to these spheres is the same as the one applied on the skybox. On the other hand, the light application for the spheres is changed in the fragment shader, where its value is the reflection vector in world coordinates with an inverted x value.

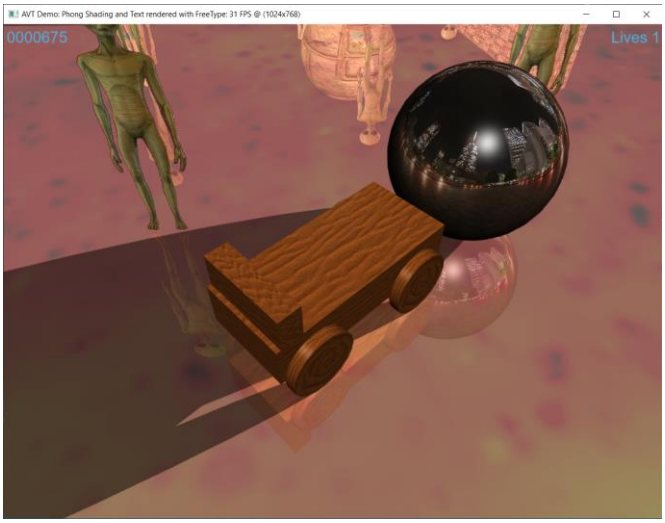

Figure 12: Sphere with applied environmental cube mapping