Band 1 (40% - 50%)

Design

The idea for this project was to model a sushi bar. A sushi bar consists of a table, some seats, and a track where the sushi moves in a circular path to serve the customers. This idea is illustrated in Figure 1.

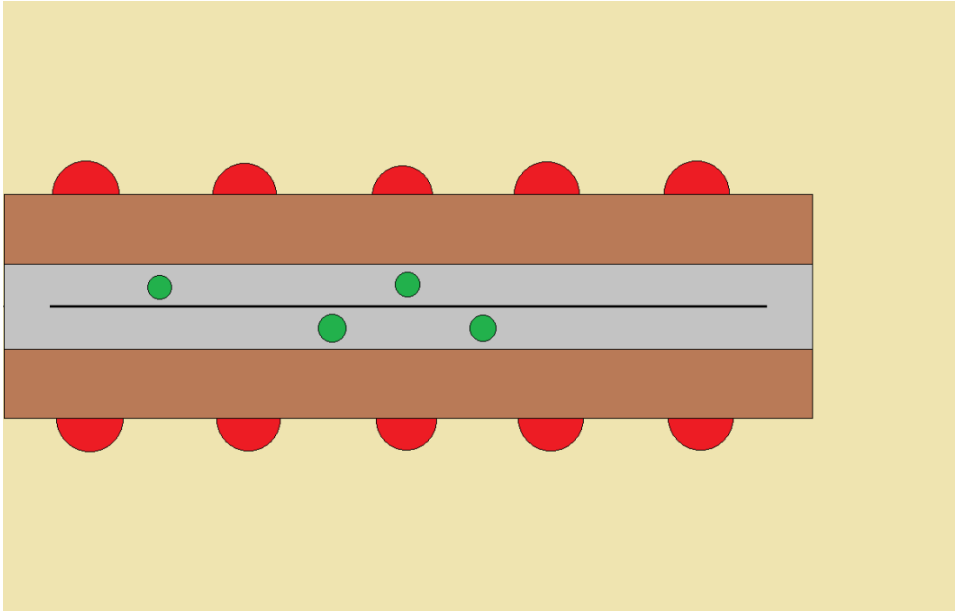

Figure 1: Basic design of the scene.

In the scene, we have a table (brown color) which hosts the track (silver color) where the sushi (green color) is rotating around the track. Several seats (red color) are placed around the table for the customers to sit.

Instancing

The table, seats and sushi are constructed using multiple simple shapes like a cylinder and cuboid. The basic shapes are constructed using functions in the shape.cpp file. Figure 2 and Figure 3 shows the code snippet for creating these shapes.

In room.cpp, we used these functions to construct more complex shapes using these basic shapes. For example, a seat is constructed using three cylinders, one for the seat, one for the base and one for connecting the seats and base together. Figure 4 shows the code snippet for generating a seat in the scene. Figure 5 shows the seats in the scene. Most of the objects which uses the functions in shape.cpp in the scene can be found in room.cpp.

```
void Shape::drawClosedCylinder(float radius, float height) {
  float y = height / 2.0f;  // Centre coordinates for cylinder

  GLUquadricObj* quad = gluNewQuadric();
  GLUquadricObj* disk1 = gluNewQuadric();
  GLUquadricObj* disk2 = gluNewQuadric();

  GLdouble slices = 12.0;
  GLdouble stack = 1.0f;

  glPushMatrix();

  // Draw hollow cylinder
  glRotatef(-90.0f, 1.0f, 0.0f, 0.0f);
  glTranslatef(0.0f, 0.0f, -y);
  gluCylinder(quad, radius, radius, height, slices, stack);

  // Draw closing disks
  gluDisk(disk1, 0, radius, slices, stack);
  glTranslatef(0.0f, 0.0f, height);
  gluDisk(disk2, 0, radius, slices, stack);

  glPopMatrix();
}
```

Figure 2: Function for creating a closed cylinder in shape.cpp.

```
void Shape::drawCuboid(float w, float h, float d, float x, float y, float z) {
  // Centralise coordinates
  x = x - (w / 2.0f);
  y = y - (h / 2.0f);
  z = z - (d / 2.0f);

  GLfloat normals[6][3] = {{ 1.0f,  0.0f,  0.0f},    // Right
                           {-1.0f,  0.0f,  0.0f},   // Left
                           { 0.0f,  0.0f,  1.0f},    // Front
                           { 0.0f,  0.0f, -1.0f},   // Back
                           { 0.0f,  1.0f,  0.0f},    // Top
                           { 0.0f, -1.0f,  0.0f}};  // Bottom

  // Right
  glNormal3fv(normals[0]);
  glBegin(GL_POLYGON);
  glVertex3f(x + w, y + 0, z + d);
  glVertex3f(x + w, y + 0, z + 0);
  glVertex3f(x + w, y + h, z + 0);
  glVertex3f(x + w, y + h, z + d);
  glEnd();

  // …

  // Bottom
  glNormal3fv(normals[5]);
  glBegin(GL_POLYGON);
  glVertex3f(x + w, y + 0, z + d);
  glVertex3f(x + w, y + 0, z + 0);
  glVertex3f(x + 0, y + 0, z + 0);
  glVertex3f(x + 0, y + 0, z + d);
  glEnd();
}
```

Figure 3: Code snippet to construct a cuboid in shape.cpp.

```
void Room::seat() {
  // Dimensions
  const float SEAT_RADIUS = 1.0f;
  const float SEAT_HEIGHT = 0.6f;

  const float ROD_RADIUS = 0.2f;
  const float ROD_HEIGHT = 3.0f;

  const float BASE_RADIUS = 0.8f;
  const float BASE_HEIGHT = 0.09f;

  Shape s;

  glPushMatrix();

  glTranslatef(0.0f, ROD_HEIGHT, 0.0f);

  // Seat
  s.drawClosedCylinder(SEAT_RADIUS, SEAT_HEIGHT);

  // Rod
  glTranslatef(0.0f, -ROD_HEIGHT / 2.0f, 0.0f);
  s.drawClosedCylinder(ROD_RADIUS, ROD_HEIGHT);

  // Base
  glTranslatef(0.0f, -ROD_HEIGHT / 2.0f, 0.0f);
  s.drawClosedCylinder(BASE_RADIUS, BASE_HEIGHT);

  glPopMatrix();
}
```

Figure 4: Code snippet for creating a seat in the scene.

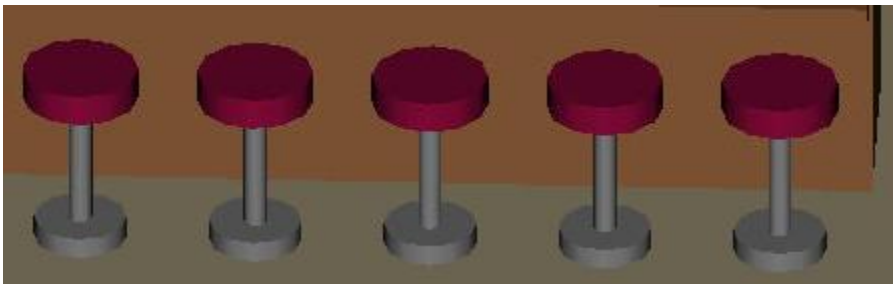

Figure 5: Seats in scene.

Lighting and material properties

Each of the object in our scene contains a material property. The material properties were sourced online from the website http://devernay.free.fr/cours/opengl/materials.html. Figure 6 shows the material properties for red plastic for the seat.

```
const Material RED_PLASTIC = {{0.00f, 0.00f, 0.00f, 1.00f},
                              {0.50f, 0.00f, 0.00f, 1.00f},
                              {0.70f, 0.60f, 0.60f, 1.00f},
                              32.00f};
```

Figure 6: Material properties for red plastic.

Another material property we have is the metal parts of the seats. When tuning the properties, the specular values for the material should be higher as metals reflect lights much better than other materials like wood. This would reflect the color of the lights more than the color of the object. Figure 7 shows the material properties for the metallic part of the seat. We used the silver material property for our metals.

```
const Material SILVER = {{0.19f, 0.19f, 0.19f, 1.00f},
                         {0.51f, 0.51f, 0.51f, 1.00f},
                         {0.51f, 0.51f, 0.51f, 1.00f},
                         51.20f};
```

Figure 7: Material properties for metallic part in the seat.

The material struct is defined in material.h but the individual material properties are set within the draw() function for each object. All the material properties that were used in the scene can be found in material.h file.

Band 2 (50% - 60%)

User interaction

The first interaction we implemented in the scene is to use the mouse to change the viewing angle of the scene. The user would need to hold left click and drag the scene to move the camera which would manipulate the viewing angle of the scene. This was implemented using the mouse event virtual functions in Qt. The function handleMouseEvent() calculates the properties of the camera. This is done in the camera.cpp file. The camera.cpp source code was adapted from Learn OpenGL https://learnopengl.com/code_viewer_gh.php?code=src/1.getting_started/7.4.camera_class/camera_class.cpp. These values are then pass onto gluLookAt() to change the viewing angle. Figure 8 shows the code snippet for adjusting the viewing angle in the scene.

```
void SushiBar::mouseMoveEvent(QMouseEvent* event) {
  if (_dragCamera) {
    QPoint mousePos = event->pos();
    float dx = mousePos.x() - _mousePos.x();
    float dy = _mousePos.y() - mousePos.y();

    _mousePos = mousePos;

    _camera->handleMouseEvent(dx, dy);
  }
}

void SushiBar::mousePressEvent(QMouseEvent* event) {
  if (event->button() == Qt::MouseButton::LeftButton) {
    _dragCamera = true;
    _mousePos = event->pos();
  }
}

void SushiBar::mouseReleaseEvent(QMouseEvent*) { _dragCamera = false; }
```

Figure 8: Code snippet to adjust viewing angle in scene using mouse events.

Band 3 (60% - 70%)

Element of animation

An element of animation in our scene is a rotating globe on the table of the sushi bar. We rotate the globe by some angle. This angle is then updated in every frame. As a result, the globe has the effect of rotating. The code snippet for rotating the globe can be found in Figure 9. Figure 10 shows the rotating globe in the scene.

```
void Globe::drawGlobe() {
  const float COORD[3] = {4.0f, -1.8f, 3.0f};

  glPushMatrix();
  glTranslatef(COORD[0], COORD[1], COORD[2]);
  glPushMatrix();
  globe();
  glPopMatrix();
  glPopMatrix();

  updateAngle();
}
```

Figure 9: Code snippet for rotating a globe.

Figure 10: The rotating globe in the scene.

Additionally, the sushi on the sushi track is also translated and rotated in the scene. Sections 1, 2, 3 and 4 represents the locations on the track. This is illustrated in Figure 11.

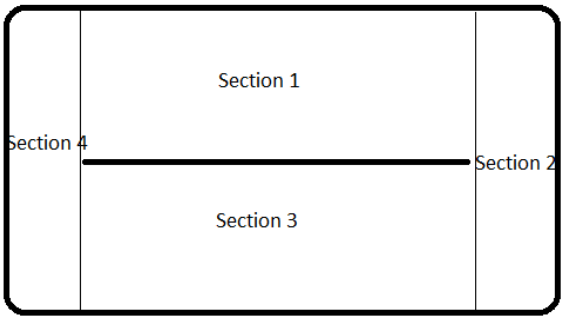

Figure 11: Illustration for the different sections on the track.

At each frame, the objects' positions are updated depending on which section they are at. When the object is at section 1 or section 3, it would just need to move accordingly on the x-axis. However, when the object reaches section 2 or 4, we would need to calculate the position with respect to the turning point of the track. This position can be calculated using trigonometric equations. The angle is updated at each frame in section 2 or 4. Figure 12 shows the code snippet for animating objects on the track.

```
void TrackObject::section1() { _position += glm::vec3(_speed, 0.0f, 0.0f); }

void TrackObject::section2() {
  const float TURNING_POINT[2] = {4.7f, 0.0f};
  if (_speed > 0) {
    float theta = _angle * glm::pi<float>() / 180.0f;
    _position.x = TURNING_POINT[0] + (_radius * glm::cos(theta));
    _position.z = TURNING_POINT[1] + (_radius * glm::sin(theta));

    _angle += _angularSpeed;
  }
}

void TrackObject::section3() { _position -= glm::vec3(_speed, 0.0f, 0.0f); }

void TrackObject::section4() {
  const float TURNING_POINT[2] = {-7.5f, 0.0f};
  if (_speed > 0) {
    float theta = _angle * glm::pi<float>() / 180.0f;
    _position.x = TURNING_POINT[0] + (_radius * glm::cos(theta));
    _position.z = TURNING_POINT[1] + (_radius * glm::sin(theta));

    _angle += _angularSpeed;
  }
}
```

Figure 12: Code snippet for moving objects on the track.

Convex object constructed from polygons

In our scene there are many convex objects constructed from polygons. For example, the track and divider are convex objects constructed from polygons. The track and the divider are created using the shape functions from Figure 3. Since a cuboid is a convex object, and the track and divider are made using a cuboid, therefore they are convex objects. The code snippet for creating a divider can be found in Figure 13. Another simple example of a convex object would be the globe as it is a sphere.

```
void Room::divider() {
  // Dimensions
  const float WIDTH = 12.0f;
  const float HEIGHT = 2.0f;
  const float DEPTH = 0.1f;
  const float COORD[3] = {-1.5f, -0.7f, 0.0f};

  glMaterialfv(GL_FRONT, GL_AMBIENT, CYAN_PLASTIC.ambient);
  glMaterialfv(GL_FRONT, GL_DIFFUSE, CYAN_PLASTIC.diffuse);
  glMaterialfv(GL_FRONT, GL_SPECULAR, CYAN_PLASTIC.specular);
  glMaterialf(GL_FRONT, GL_SHININESS, CYAN_PLASTIC.shininess);

  Shape s;
  s.drawCuboid(WIDTH, HEIGHT, DEPTH, COORD[0], COORD[1], COORD[2]);
}
```

Figure 13: Code snippet for creating the track divider.

Textures

The globe uses the Mercator-projection.ppm texture file. The texture is mapped onto the sphere. We begin by creating a reference to a texture. Then we bind the image to the texture reference. Next, we set the texture parameters for the texture. Finally, we map the texture onto the object. The code snippet for mapping the texture onto the globe can be found in Figure 14. This creates the globe from Figure 10.

```
  glEnable(GL_TEXTURE_2D);
  unsigned int _tex;
  glGenTextures(1, &_tex);
  glBindTexture(GL_TEXTURE_2D, _tex);

  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, _image->width(), _image->height(), 0,
               GL_RGB, GL_UNSIGNED_BYTE, _image->data());

  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

  gluQuadricDrawStyle(quad, GLU_FILL);
  gluQuadricTexture(quad, GL_TRUE);
  gluQuadricNormals(quad, GLU_SMOOTH);
```

Figure 14: Code snippet for mapping texture onto the globe.

The ppm files for Marc and Markus were used as pictures and were placed onto the walls of the sushi bar. The code snippet for creating the pictures can be found in Figure 15. Figure 16 shows the pictures in

the scene. For some reason, I couldn't get Markus's photo to render properly. Other textures however are mapped correctly.

```
glEnable(GL_TEXTURE_2D);

glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);

// Load texture
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, image->width(), image->height(), 0,
             GL_RGB, GL_UNSIGNED_BYTE, image->data());

// Texture settings
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

glBindTexture(GL_TEXTURE_2D, texture);

// Frame

glBegin(GL_QUADS);
glNormal3f(0.0f, 0.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f);  glVertex3f( 1.0f, -1.0f, 0.0f);
glTexCoord2f(0.0f, 0.0f);  glVertex3f(-1.0f, -1.0f, 0.0f);
glTexCoord2f(0.0f, 1.0f);  glVertex3f(-1.0f,  1.0f, 0.0f);
glTexCoord2f(1.0f, 1.0f);  glVertex3f( 1.0f,  1.0f, 0.0f);
glEnd();

// Unbind texture
glBindTexture(GL_TEXTURE_2D, 0);
glDisable(GL_TEXTURE_2D);
```

Figure 15: Code snippet for generating picture.

Figure 16: Pictures on the wall.

Band 4 (70% - 100%)

Hierarchical modelling

In our scene, we have a lucky cat that sits on the sushi track. The right hand of the cat is constantly oscillating from a vertical position to a horizontal position. The cat sits on the track which is moving in a circular path though the scene. Despite loading the model, the creator of the model did not share a texture file hence I could not implement texture mapping for the lucky cat. Therefore, I set the color of the body and the hand in two distinct colors to make the movement more obvious. The animation of the cat is independent from the animation where it moves along the track. The animation for the hand is animated in luckycat.cpp. Figure 17 shows the code snippet for moving the cat's hand. As for the animation for moving the cat along the track, this is shown in Figure 12. As a result, the cat is rendered like in Figure 18. The model of the cat can be found in https://www.cgtrader.com/free-3d-models/furniture/other/maneki-neko-5c986741-0416-4673-9dcf-8224a67cacde.

```
// Draw body
glRotatef(-_angle + 90.0f, 0.0f, 1.0f, 0.0f);
glPushMatrix();
glScalef(SCALE, SCALE, SCALE);
drawBody();

// Draw hand
float angle =
    45.0f * glm::sin(_handAngle * glm::pi<float>() / 180.0f) + 45.0f;
glTranslatef(0.0f, 7.0f, 0.0f);      // Offset to hand position
glRotatef(angle, 1.0f, 0.0f, 0.0f);  // Perform hand rotation
glTranslatef(0.0f, -7.0f, 0.0f);     // Return back to original position

drawHand();
```

Figure 17: Code snippet for moving hand.

Figure 18: Lucky cat moving its hands.

Additional user interactions

In our scene, we have a slider that controls the speed of the track. The track can be adjusted into 9 different speeds. We used Qt's signal and slots to implement changing the speed of the track. The code for creating a slider and adjusting the speed can be found in Figure 19.

```
void Window::trackSpeedSlider() {
  trackSpeed = new QSlider(Qt::Horizontal);

  trackSpeed->setRange(1, 10);
  trackSpeed->setSingleStep(1);

  for (Sushi* sushi : sushiBar->_sushis)
    connect(trackSpeed, SIGNAL(valueChanged(int)), sushi, SLOT(setSpeed(int)));

  connect(trackSpeed, SIGNAL(valueChanged(int)), sushiBar->_cat,
          SLOT(setSpeed(int)));

  windowLayout->addWidget(trackSpeed);
}


void TrackObject::setSpeed(int speed) {
  _speed = (float)speed * 0.05f;
  _angularSpeed = _speed / _radius * 180.0f / glm::pi<float>();
}
```

Figure 19: Code snippet for changing the track speed using slider.

Additionally, we have a button that allows the user to pause the track. Pressing the button again will resume the track. Similarly, we used Qt's signal and slot to pause and play the track. The code for creating the pause button can be found in Figure 20.

```
void Window::pauseTrackButton() {
  pauseTrack = new QPushButton("Pause Track");

  for (Sushi* sushi : sushiBar->_sushis)
    connect(pauseTrack, SIGNAL(pressed()), sushi, SLOT(stopTrack()));

  connect(pauseTrack, SIGNAL(pressed()), sushiBar->_cat, SLOT(stopTrack()));

  windowLayout->addWidget(pauseTrack);
}

void TrackObject::stopTrack() {
  if (_speed > 0.0f) {
    _oldSpeed = _speed;
    _oldAngularSpeed = _angularSpeed;
    _speed = 0.0f;
  } else {
    _speed = _oldSpeed;
    _angularSpeed = _oldAngularSpeed;
  }
}
```

Figure 20: Code snippet for creating pause button.

Outcome

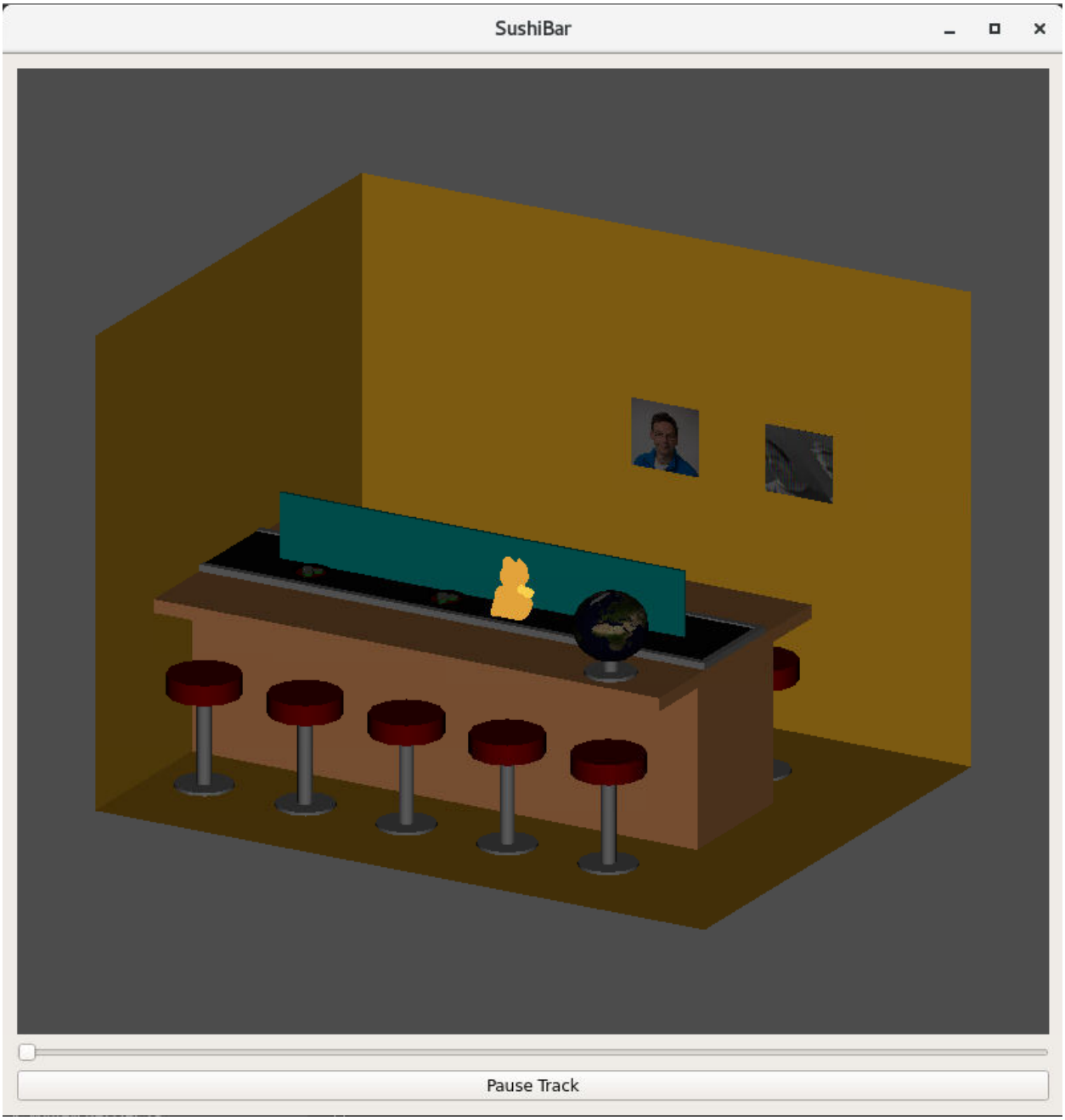

Figure 21: Outcome of the scene.