ARENA:

Completed up to HARD (18 points)

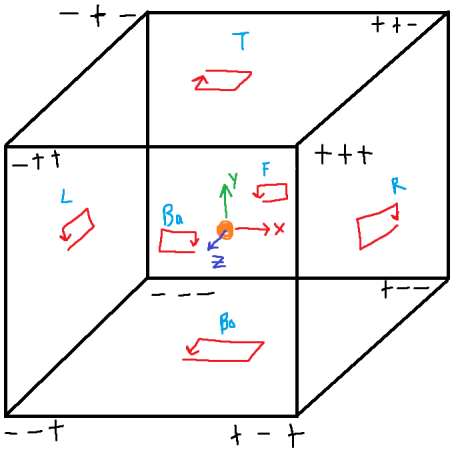
If there is a collision with a wall and the collision radius of the ship (`GameManager::handleWallCollisions`, `collision::withWall`), the game is reset. If there is a collision with a wall and the warning radius of the ship, the colour is set to red. Otherwise, the colour is set to white.

At the start of every display call (`GameManager::onDisplay`), after setting the model view matrix to the identity, the camera is rotated (`Camera::rotate`), the skybox is drawn (`Skybox::draw`) without depth testing (the first draw call is shown), and the camera is translated (`Camera::translate`).

Since all other entities are drawn after the skybox with depth testing turned on, they appear above the skybox. Since camera translation occurs last, the skybox can be drawn as a `1x1x1` cube in the center of the arena rather than having to move it with the ship. Finally, camera rotation occurs separately beforehand, meaning that the skybox can remain stationary at the origin while the ship is rotated. Together, this lets the skybox appear infinitely far away, and remains "stationary" while the ship flies and rotates around the arena.

Originally I tried to separately rotate and translate the camera using glLookAt, but ultimately decided to manually translate the camera around the world and use a quaternion to handle rotations. This proved invaluable, since giving the camera a quaternion meant that bullet billboarding was trivial.

As discussed in our emails, drawing the skybox was initially difficult because of OpenGL expecting image binary streams to start from the bottom left pixel rather than the top left pixel, which was fixed with `stb_flip_image_on_load(true)`.



SPACESHIP MODEL:

Completed up to MEDIUM (10 points, running total: 28)

The spaceship model is taken from here and is licensed under CC Attribution.

To make drawing easier, the OBJ was triangulated using Blender so that it would be guaranteed that each face is made up of 3 vertices. The OBJ was loaded using tinyobjloader. Vertices, texture coordinates and normals (VTNs) were converted from flat arrays of floats to Vectors. Since tinyobj provides face indices based on the entire group of vertices defining the VTNs, grouping each as a vector made their access while drawing much easier.

Adapting the ASCII explanation from tinyobjloader's GitHub:

```
Ship::vertices, Ship::uvs, Ship::normals => Vector3Ds. UVs do not have a Z coord, i.e. Vector3D(x,y,0)

        v[0]              v[1]              v[2]                   v[n-1]
  +-----------------+-----------------+-----------------+     +-----------------+
  | Vector3D(x,y,z) | Vector3D(x,y,z) | Vector3D(x,y,z) | ... | Vector3D(x,y,z) |
  +-----------------+-----------------+-----------------+     +-----------------+


Ship::triangles => Three arrays of size 3, holding the index of the VTNs which make up the face, and the
material ID (m) that face uses. For example, this is how the vertices of the first face are mapped:

For each of the ship's triangles:
            t[0]                    t[1]                    t[2]                         t[n-1]
  +----------------------+----------------------+----------------------+     +----------------------+
  | Triangle(v, t, n, m) | Triangle(v, t, n, m) | Triangle(v, t, n, m) | ... | Triangle(v, t, n, m) |
  +----------------------+----------------------+----------------------+     +----------------------+
      |
      |/
    t[0].vertices[0]   t[0].vertices[1]   t[0].vertices[2]
  +------------------+------------------+------------------+
  |       342        |       138        |        198       |
  +------------------+------------------+------------------+
           |                  |                   |
           +------------------|-------------------|--------+
           +------------------+     +-------------+        |
           |/                       |/                     |/
        v[138]                   v[198]                 v[342]
  +-----------------+     +-----------------+     +-----------------+
  | Vector3D(x,y,z) | ... | Vector3D(x,y,z) | ... | Vector3D(x,y,z) |
  +-----------------+     +-----------------+     +-----------------+


These are the three vertices which are drawn for this face. Texture coordinates and normal coordinates are
fetched similarly.


Ship::materials => Holds std::array<float, 4> for each material in the material file. Like faces, material
IDs map to the corresponding position in the material vector.
```

Hard: Not implemented. I could have easily implemented this if I separated the ship into different shapes in Blender (e.g. making the wings a separate joint object), but I REALLY need to put this assignment down and work on other stuff. *Really.*

ASTEROID MODEL:

Completed up to MEDIUM (12 points, running total: 40)

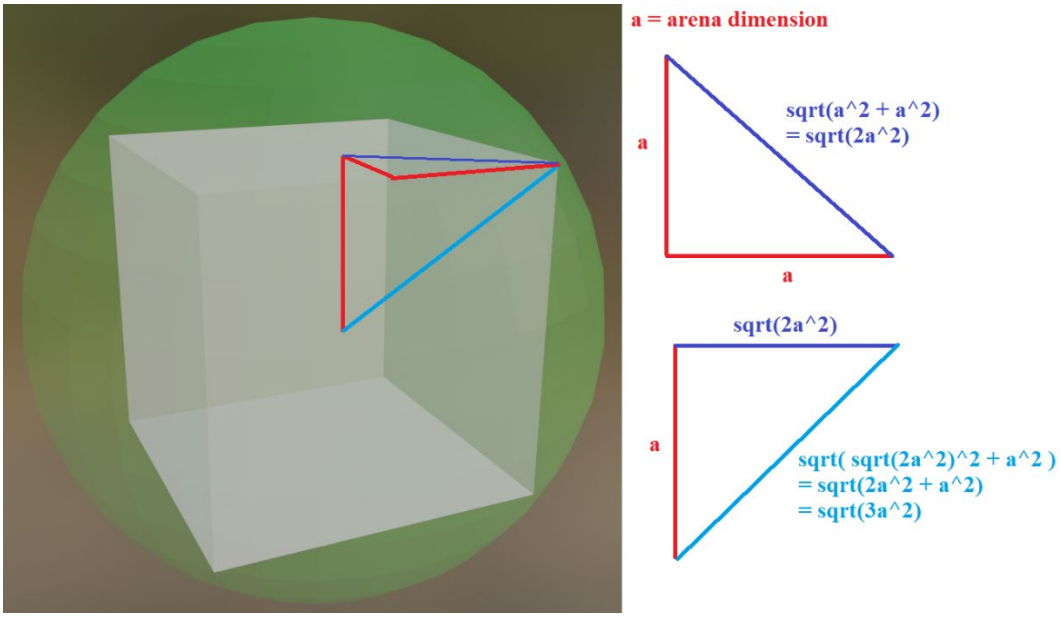
Not sure if there is much to say here. I basically took your code, slapped the texture you gave us onto it and bam, asteroids.

Asteroids were perturbed on the X/Z plane by just increasing/decreasing by a small random float.

ASTEROID MOVEMENT:

Completed up to HARD (14 points, running total: 54)

Asteroids spawn at a random point on a sphere bounding the arena. Assuming that the arena is a cube, the radius of this sphere is equal to $\sqrt{3 \times (arena\ dimension)^2}$ (see diagram below). Asteroids rotate about a random rotation axis. Asteroids are generated with a random rotation speed and rotation direction. On update, its rotation angle is simply updated.



Asteroids bounce off of walls by checking which wall they have collided with and reversing the appropriate component of their velocity:

```
void collision::resolve(const Wall& wall, Asteroid& asteroid) {
    if (wall.getSide() == Side::TOP || wall.getSide() == Side::BOTTOM) {
        asteroid.reverseY();
    }
    else if (wall.getSide() == Side::LEFT || wall.getSide() == Side::RIGHT) {
        asteroid.reverseX();
    }
    else if (wall.getSide() == Side::FRONT || wall.getSide() == Side::BACK) {
        asteroid.reverseZ();
    }
}
```

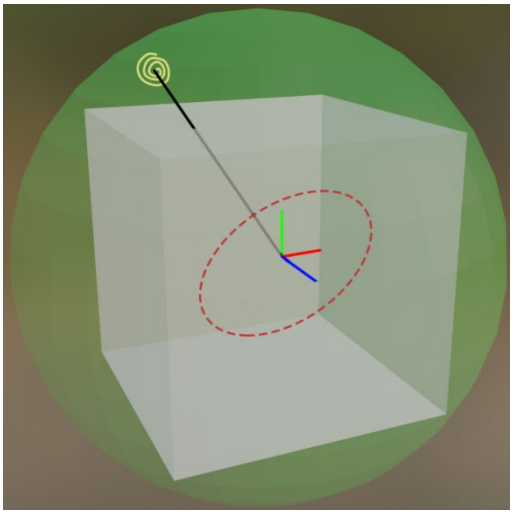
The radius of an asteroid is randomly selected from a range of floats. Its mass is simply its volume. Asteroids bounce off of each other using the equation given at the bottom of [the Wikipedia article on elastic collisions]():

$$\mathbf{v}_1' = \mathbf{v}_1 - \frac{2m_2}{m_1 + m_2} \frac{\langle \mathbf{v}_1 - \mathbf{v}_2, \mathbf{x}_1 - \mathbf{x}_2 \rangle}{\|\mathbf{x}_1 - \mathbf{x}_2\|^2} (\mathbf{x}_1 - \mathbf{x}_2),$$
$$\mathbf{v}_2' = \mathbf{v}_2 - \frac{2m_1}{m_1 + m_2} \frac{\langle \mathbf{v}_2 - \mathbf{v}_1, \mathbf{x}_2 - \mathbf{x}_1 \rangle}{\|\mathbf{x}_2 - \mathbf{x}_1\|^2} (\mathbf{x}_2 - \mathbf{x}_1)$$

Even though the article says these equations only work in 2D, they also (for some reason) work in 3D! When two asteroids collide, their new velocities are determined according to the above equation. Then, each asteroid is explicitly updated once such that they move slightly away from each other so that multiple collisions do not occur.

```
// ASTEROID->ASTEROID COLLISIONS ////////////////////////
for (Asteroid& a2 : asteroid_field->getAsteroids()) {
    if (a1.id() != a2.id()) {
        if (collision::withAsteroid(a1.getPosition(), a1.getRadius(), a2.getPosition(), a2.getRadius())) {
            // Calculate new velocities, then move slightly apart
            collision::resolve(a1, a2);
            a1.update(dt);
            a2.update(dt);
        }
    }
}
```

LIGHTING:



Completed up to HARD (6 points, running total: 60)

GL_LIGHT_0 is a directional light, with position `{ 1.0, 0.0, 0.0, 0.0 };`. Since the last value is 0, this is a directional light pointing in the positive x-axis.

GL_LIGHT_1 is an animated light (a “satellite”). When the game is started, a random point on a bounding sphere around the arena is chosen as the starting position of the satellite (black). Its axis of orbit is calculated by generating a random unit vector at the origin of the world and taking the cross product with the position vector. By definition of the cross product, this produces a vector on the plane perpendicular to the position of the satellite (red). This vector is used as the rotation axis in the corresponding draw function.

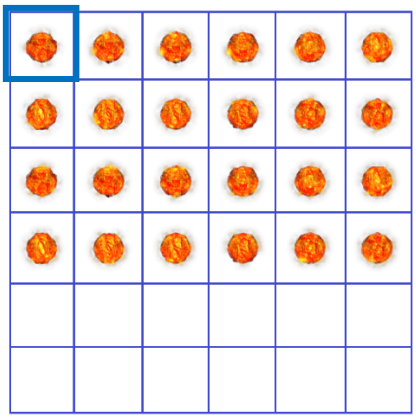
BULLETS AND SHOOTING:

Completed up to HARD (12 points, running total: 72)

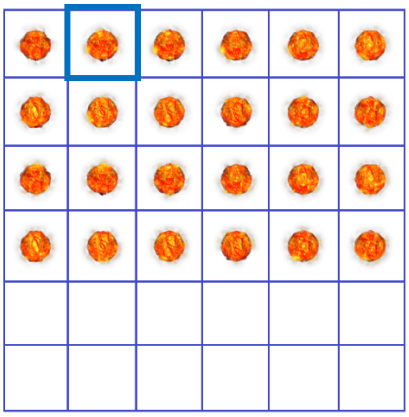
The camera has a static method to grab its current rotation (a quaternion). When a bullet is drawn, its quad is rotated using `glMultMatrixf(Quaternion::toMatrix(Camera::getRotation()).data())`. This ensures that bullet textures are always facing the camera.

Animated objects, including bullets, are drawn with the AnimationDrawer class. Each animated object has its own drawer. To animate the bullets, a `std::vector<std::vector<std::pair<float, float>>>` is created. Each element of the array contains an `std::pair<float, float>` , corresponding to the (u, v) coordinates for that given position on the texture map.

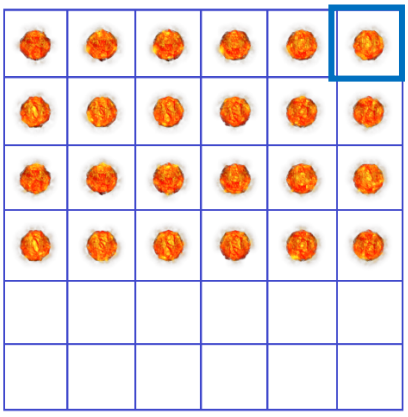
To draw a texture, a sliding 2x2 window is used. The corners of the window correspond to the four UV coordinates that will be drawn on the quad for that frame. The frame moves from top to bottom, left to right.



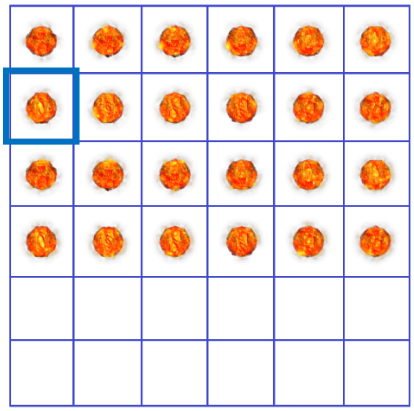
The window begins at the top left of the texture map



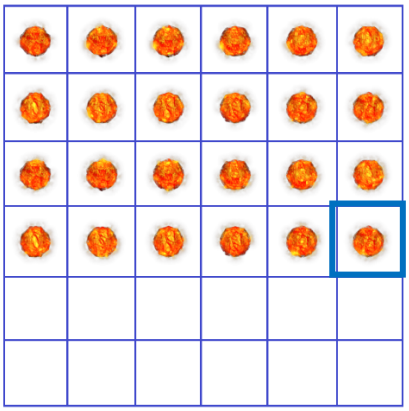
When the next frame is due, the window moves one position right



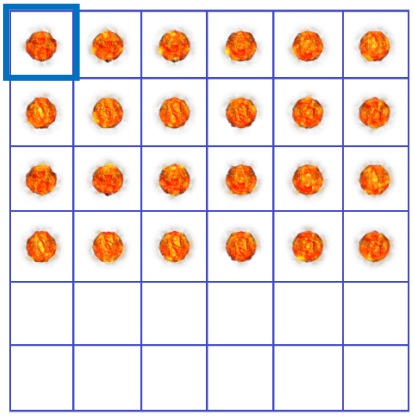
Eventually the frame will reach the end of the current row



The window moves back to the left side and goes down one row



Eventually the window reaches the last bullet texture



The window resets to the starting position and the process repeats

Bullets inherit from the class `Transparent`, which dictates that the methods `Transparent::draw()` and `Transparent::getPosition()` must be overridden. All transparent object drawing is handled by the static `Transparent::drawAll()` method. Whenever a transparent object is created, it is added to a static vector in the `Transparent` class. Before drawing, that vector is sorted using `std::sort()` in descending order of distance from the camera using a lambda comparator.

One problem which could not be overcome was Z-fighting. Despite bullets being successfully sorted as above, bullets that are similar distances from the camera nevertheless render their transparent regions over one another. This is not noticeable at high bullet velocities.

EXPLOSIONS:

Completed up to HARD (8 points, running total: 80)

Explosion texture map source:

https://www.nicepng.com/ourpic/u2q8y3r5i1t4w7q8_explosion-texture-png-explosion-texture-png-emoji-african/

Draw code is analogous to what is shown in BULLETS AND SHOOTING, except the window moves left to right, top to bottom. Explosions have random velocities, decay at the same rate, and delete themselves after a full cycle, unlike bullets which loop until collision.

One weakness of the current architecture of my assignment is that each class having an AnimationDrawer means the UV vectors are remade on creation of every object. There was probably a way to fix this to only occur once but I decided not to dedicate time to the task.

CAMERA AND SHIP MOVEMENT:

Completed up to HARD (18 points, running total: 98)

Instead of using glutLookAt, the world “camera” is simply a translation and rotation of the world according to the camera position and camera rotation (a quaternion), respectively.

If none of the camera movement keys are pressed, the camera is given a default Look::AHEAD state, meaning its position is relative to the ship, but its rotation is simply the ship’s rotation. For any other state, the rotation is multiplied by the quaternion constructed from the appropriate axis and rotation angle to have it face the correct direction given the key pressed. Likewise, the position is rotated in a circle around the ship in the appropriate axis.

```
void GameManager::updateCamera() {
    Vector3D position;
    Quaternion rotation;

    if (camera->look_at == Look::AHEAD) {
        rotation = ship->getRotation();
        position = ship->getPosition() + camera->distanceFromShip() * (ship->getRotation() * Vector3D::forward());
    }
    else if (camera->look_at == Look::LEFT) {
        rotation = ship->getRotation() * Quaternion(Vector3D::up(), -90);
        position = ship->getPosition() + camera->distanceFromShip() * (ship->getRotation() * Vector3D::right());
    }
    else if (camera->look_at == Look::RIGHT) {
        rotation = ship->getRotation() * Quaternion(Vector3D::up(), 90);
        position = ship->getPosition() - camera->distanceFromShip() * (ship->getRotation() * Vector3D::right());
    }
    else if (camera->look_at == Look::BEHIND) {
        rotation = ship->getRotation() * Quaternion(Vector3D::up(), 180);
        position = ship->getPosition() - camera->distanceFromShip() * (ship->getRotation() * Vector3D::forward());
    }
    else if (camera->look_at == Look::ABOVE) {
        rotation = ship->getRotation() * Quaternion(Vector3D::right(), -90);
        position = ship->getPosition() - camera->distanceFromShip() * (ship->getRotation() * Vector3D::up());
    }
    else if (camera->look_at == Look::BELOW) {
        rotation = ship->getRotation() * Quaternion(Vector3D::right(), 90);
        position = ship->getPosition() + camera->distanceFromShip() * (ship->getRotation() * Vector3D::up());
    }

    // raise the position of the camera slightly up
    position += 10.0f * (ship->getRotation() * Vector3D::up());

    camera->lerpPositionTo(position);
    camera->lerpRotationTo(rotation);
```

For smooth camera movement, the position and rotations are LERPed and SLERPed, respectively.

```
void Camera::lerpPositionTo(Vector3D new_position) {
    position = Vector3D::lerp(position, new_position, CAMERA_LERP_T);
}

void Camera::lerpRotationTo(Quaternion new_rotation) {
    rotation = Quaternion::slerp(rotation, new_rotation, CAMERA_LERP_T);
}
```