COSC363 Assignment 1

Cameron Stevenson
57052612

Compilation/Running

I've included a VM lab box compiled labcompile.out, so run *./labcompile.out* . If that doesn't work try run *make* and then *./assignment1.out*

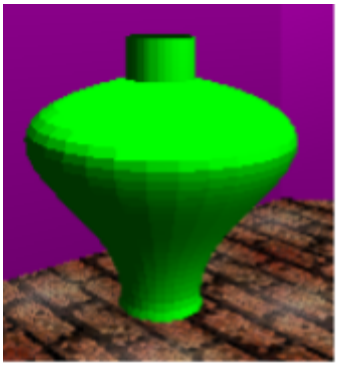
Controls

Up Arrow Key - move forward
Down Arrow Key - move back
Left Arrow Key - turn left
Right Arrow Key - turn right

Models

Model 1 - Vase[1]

This is a surface of revolution, generated by $radius = max(0.4,\ 0.6 + 0.4(y+1)^2 - e^{y-1})$ (Figure 1.1). Confined to $-1 < y < 3.5$, this gives the vase curve. To develop this equation I started with $max(0.4,\ ...)$ as a way of getting a straight neck once the curve dropped off below 0.4. The curve starts with an increasing quadratic for the slowly expanding base, which is then suppressed by the negative exponential



Model 1 - Vase

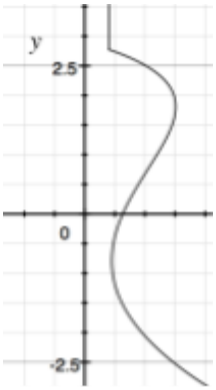


Figure 1.1

```
Vector3f calculateVertex(float verticalSegmentSize, float radialSegmentSize, float v, float r) {
    float y = -1 + v * verticalSegmentSize;
    float angle = r * radialSegmentSize;
    float radius = math::max(0.4, 0.6 + 0.4 * (y + 1) * (y + 1) - math::exp(y - 1));
    float x = radius * math::sinrad(angle);
    float z = radius * math::cosrad(angle);
    return Vector3f(x, y, z);
}
for(int v = 0; v <= verticalSubdivisions; v++) {
    for(int r = 0; r < radialSubdivisions; r++) {
        Vector3f vec3 = calculateVertex(verticalSegmentSize, radialSegmentSize, v, r);
        _vertices.push_back(vec3);
    }
}
```

Figure 1.2

[1] I hadn't seen the vase in lab 4 before making this one

```
for(int v = 0; v < verticalSubdivisions; v++) {
    for(int r = 0; r < radialSubdivisions; r++) {
        vector<int> tri1 = vector<int>(3);
        tri1[0] = v * radialSubdivisions + r; //bottom left
        tri1[1] = v * radialSubdivisions + (r + 1) % radialSubdivisions; //bottom right
        tri1[2] = (v + 1) * radialSubdivisions + r; //top left
        vector<int> tri2 = vector<int>(3);
        tri2[0] = v * radialSubdivisions + (r + 1) % radialSubdivisions; //bottom right
        tri2[1] = (v + 1) * radialSubdivisions + (r + 1) % radialSubdivisions; //top right
        tri2[2] = (v + 1) * radialSubdivisions + r; //top left

        _tris.push_back(tri1);
        _tris.push_back(tri2);
```

Figure 1.3

```
Vector3f u1 = _vertices[tri1[1]] - _vertices[tri1[0]];
Vector3f v1 = _vertices[tri1[2]] - _vertices[tri1[0]];
Vector3f normal1 = Vector3f::cross(u1, v1);
_normals.push_back(normal1);

Vector3f u2 = _vertices[tri2[1]] - _vertices[tri2[0]];
Vector3f v2 = _vertices[tri2[2]] - _vertices[tri2[0]];
Vector3f normal2 = Vector3f::cross(u2, v2);
_normals.push_back(normal2);
```

Figure 1.4

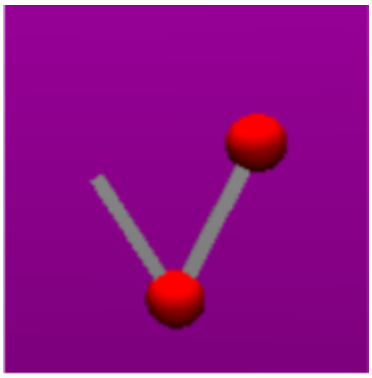
term for the rapidly shrinking top. The vase vertices are generated in slices of constant y, then by angle around the y-axis (Figure 1.2). Faces are made as triangles between adjacent vertices in the same slice and adjacent slices (Figure 1.3). Normals are made per-triangle (Figure 1.4).

Model 2 - Rubix Cube

Model 2 - Rubix Cube

Each individual block is first drawn in solid colors for the faces, and then in black wireframe for the distinctive borders between the blocks. The cube keeps track of a rest state (not in the middle of rotating a face), and a face rotation. Each block tracks its own position in the rest state. Only the overall cube and the drawing code is aware of the rotation, until the rotation is complete. Then the blocks update to their new rest state positions, and rotate their own face colours according to the face rotation. The cube reconciles where each block is (this keeps an ordered array of blocks according to position in the rest state, to make animation calculations less expensive, as those are called per frame). I used this updating rest state + current turn method to avoid storing a long history of rotations from the very start.

Model 3 - Chaotic Double Pendulum



Model 3 - Double Pendulum

This is a physics simulation using the motion equations (Figure 3.1) developed at MIT[2], and Euler's method. Simulating updates

[2] https://web.mit.edu/jorloff/www/chaosTalk/double-pendulum/double-pendulum-en.html

every frame was not often enough, and resulted in large energy losses whenever the pendulum hit a very rigid bounce. To counter this I simulated updates at a higher frequency (Figure 3.2). This was successful, but energy was still being lost slowly over time. I tried various methods of injecting energy back into the system. I started by trying to calculate what the angular velocity of the outside pendulum should be to return to the initial energy, but this was unsuccessful. I tried damping/accelerating one pendulum at a time, but due to how the pendulum's affect each other, sometimes this would have an opposite effect on the other pendulum and it would hit a steady state of uncontrollably increasing energy. I finally found that damping/accelerating both pendulums simultaneously (Figure 3.3) would keep conservation of energy in the system, without tending towards any obvious steady states.

$$\omega_1' = \frac{-g\,(2\,m_1 + m_2)\sin\theta_1 - m_2\,g\sin(\theta_1 - 2\,\theta_2) - 2\sin(\theta_1 - \theta_2)\,m_2\,(\omega_2^{\,2}\,L_2 + \omega_1^{\,2}\,L_1\cos(\theta_1 - \theta_2))}{L_1\,(2\,m_1 + m_2 - m_2\cos(2\,\theta_1 - 2\,\theta_2))}$$

$$\omega_2' = \frac{2\sin(\theta_1 - \theta_2)\,(\omega_1^{\,2}\,L_1\,(m_1 + m_2) + g(m_1 + m_2)\cos\theta_1 + \omega_2^{\,2}\,L_2\,m_2\cos(\theta_1 - \theta_2))}{L_2\,(2\,m_1 + m_2 - m_2\cos(2\,\theta_1 - 2\,\theta_2))}$$

Figure 3.1

```
int res = 100;
for(int i = 0; i < res; i++) {
    doublePendulum_.update((dt / res));
}
```

Figure 3.2

```
if(energy() > initialEnergy) {
    w1 *= 0.99;
    w2 *= 0.99;
}
if(energy() < initialEnergy) {
    w1 *= 1.01;
    w2 *= 1.01;
}
```

Figure 3.3

Extra Features

Spotlight on a rotating object (Figure 4.1) - the spotlight in the top middle of the room, rotates back and forth, its beam is visible on the floor and the vase

Physics based simulation - see Model 3 - Chaotic Double Pendulum for the description

A surface shape generated using a mathematical formula - see Model 1 - Vase for the description

Skybox - simple cube with 6 2D textures - textures sourced from Humus[3]

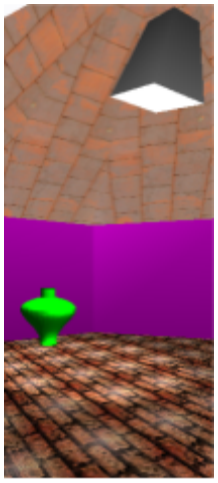


Figure 4.1

[3] http://www.humus.name/index.php?page=Cubemap&item=Yokohama