CS 432 – Interactive Computer Graphics

Lecture 09 – Part 2

Particles

Topics

- Particles

Reading

- Angel
  - Particles: 9.2-9.6

Links

- http://ogldev.atspace.co.uk/www/tutorial28/tutorial28.html
- http://www.swiftless.com/tutorials/opengl/particles.html

Particle Systems

- Particles are an important *procedural* method
  - Updated based on functions, rules, mathematics, etc..
- Particles can be used to model
  - Natural phenomena
    - Clouds
    - Terrain
    - Plants
  - Crowd Scenes
  - Real Physical Processes

Newtonian Particle

- Often (as therefore as a good example) we consider a particle system that is subject to Newton's laws
- Each particle
  - Is an ideal point mass, m
  - Has six degrees of freedom
    - Position
    - Velocity
  - Obeys Newton's laws
    - $f = ma$

Newtonian Particle

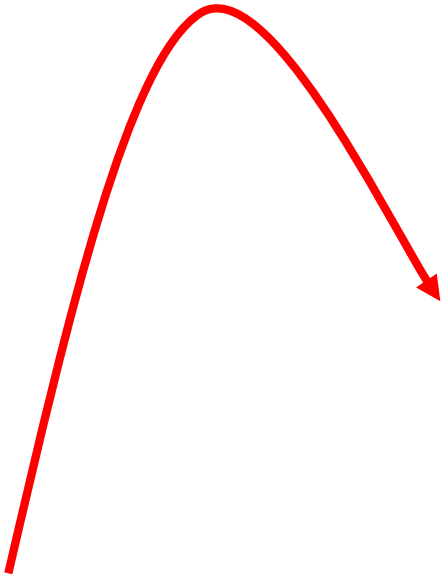
- Position
  - $p_i = (x_i, y_i, z_i)$
- Velocity = change in position
  - $v_i = \frac{dp_i}{dt} = (\frac{dx_i}{dt}, \frac{dy_i}{dt}, \frac{dz_i}{dt})$
- Force = mass*acceleration
  - Acceleration = change in velocity
  - $f_i = m\left(\frac{dv_i}{dt}\right)$
- Hard part is defining force vector

Force Vector

- Independent Particles
  - Examples forces
    - Gravity
    - Wind forces
  - Take $O(n)$ calculations
- Locally Coupled Particles $O(n)$
  - Meshes
  - Spring-Mass Systems
- Globally Coupled Particles $O(n^2)$
    - Attractive and repulsive forces

Independent Particles/Simple Forces

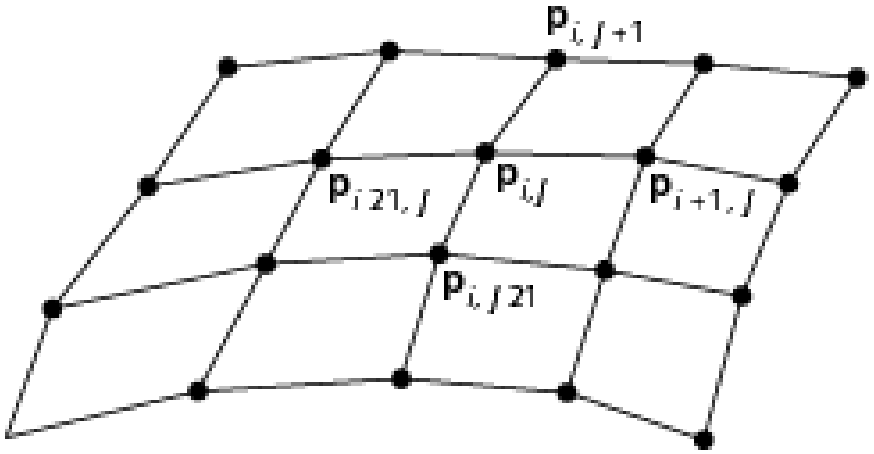
- Consider force on particle i
  - $f_i = f(p_i, v_i)$
    - The force may a function of current position and velocity
- Example: Gravity $f = g$
  - $f = (0, -g, 0)$
- Others
  - Wind forces
  - Drag



$\mathbf{p}_i(t_0), \mathbf{v}_i(t_0)$

Locally Coupled Particles

- One example of *locally coupled particles* is a **mesh**
- Connect each particle to its closest neighbors
  - O(n) force calculations
    - Each of the n particles takes into account maybe 4 neighbors
- A force that behaves like this is a *spring force*



Spring Forces

- Assuming each particle has unit mass and is connected to its neighbor(s) by a spring
- Hooke's law: force proportional to distance, $d = ||p - q||$, between points

Hooke's Law

- Let
  - s be the distance when there is no force (the length of the spring when at rest)
  - k_s be the spring constant
  - $\frac{d}{|d|}$ is a unit vector pointed from p to q
- Then we can compute the spring force as

$$f = -k_s(|d| - s)\frac{d}{|d|}$$

NOTE: Each interior point in mesh has four forces applied to it

Attraction and Repulsion

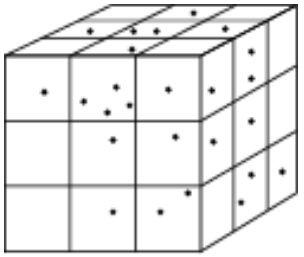
- We can also add/use attractive and/or repulsive forces
- Repulsion
  - Maybe $f = -k_r \frac{d}{|d|^3}$
  - Where k is some coefficient
- Gravitational attraction $k = Gm_a m_b$ so
  - $f = G \frac{m_a m_b}{|d|^2} \frac{d}{|d|}$
- General case requires $O(n^2)$ calculations
  - Every particle must take every other one into account
- In most problems, the drop off is such that not many particles contribute to the forces of any given particle

Global Particle Systems

- The attraction and repulsive “free” systems allow for each particle to act on each other particle
  - To have every particle effect every other particle has complexity $O(n^2)$
- We may want to *approximate* global effects by constraining what particles should effect which particles
  - In most problems, the drop off is such that not many particles contribute to the forces of any given particle
- Approaches include:
  - Having a linked-list of nearest neighbors for each particle
  - Only allow particles within some area to effect a given particle

Boxes

- Spatial subdivision technique
- Divide space into boxes
- Particle can only interact with particles in its box or the neighboring boxes
- Must update which box a particles belongs to after each time step



Linked Lists

- Each particle maintains a linked list of its neighbors
- Update data structure at each time step

Simple Particle System

- Let's create structures for our particles
- ```
  struct particle{
      vec4 color;
      vec4 position;
      vec4 velocity;
      float mass;
  };
  ```
- And create a particle system as an array of particles
  - `particle particles[MAX_NUM_PARTICLES];`
  ```

Initializing Particle System

- Let's initialize them with random locations inside a centered cube with side length 2.0 and with random velocities

```
//Particles
const int NUM_PARTICLES = 1000;
particle particles[NUM_PARTICLES];
vec4 particlePoints[NUM_PARTICLES];
vec4 particleColors[NUM_PARTICLES];
void initializeParticles();
void updateParticles();
void drawParticles();
float applyForces(int,int);
void testCollision(int);
```

```
void initializeParticles(){
    for(int i = 0; i < NUM_PARTICLES; i++){
        particles[i].mass = 1.0;
        for(int j=0; j < 3; j++){
            particles[i].color[j]=(float)rand()/RAND_MAX;
            particles[i].position[j] = 2.0*((float)rand()/RAND_MAX)-1.0;
            particles[i].velocity[j] = 2.0*((float)rand()/RAND_MAX)-1.0;
        }
        particles[i].color.w = 1.0;
        particles[i].position.w = 1.0;
        particles[i].velocity.w = 0.0;
    }
}
```

Updating Particle Positions (CPU)

- We can either use the idle or the timer callback to update positions

```
float last_time, present_time;
void idle(){
    float dt;
    present_time = glutGet(GLUT_ELAPSED_TIME);
    dt = 0.001*(present_time-last_time);
    for(int i=0; i<num_particles;i++){
        for(int j=0;j<3;j++){
            particles[i].position[j]+= dt*particles[i].velocity[j];
            particles[i].velocity[j]+= dt*forces(i,j)/particles[i].mass;
        }
        collision(i);
    }
    last_time = present_time;
    glutPostRedisplay();
}
```

Update the j^{th} component of the velocity vector based on forces

$$f = ma$$

$$a = f/m$$

Forces

- For this simple system let's just use gravity
  - We could also use attractive/repulsive forces

```
float forces(int i, int j){
    if(j==1) //only affect y direction
          return -1;
    else
          return 0;
}
```

Collisions

- Lets keep the particles within the length-2 box
- So we need to check to see if particle's step will cause it to cross a side of the box
  - If it does, we will bounce as a reflection

```
float coef;  //how strong it bounces back
void collision(int n){
    for(int i=0;i<3; i++){
            if(particles[n].position[i] >=1.0){
               particles[n].velocity[i] *= -coef;
               particles[n].position[i] = 1.0-
                      coef*(particles[n].position[i]-1.0);
            }
            if(particles[n].position[i]<=-1.0){
               particles[n].velocity[i] *= -coef;
               particles[n].position[i] = -1.0-
                      coef*(particles[n].position[i]+1.0);
            }
    }
}
```

Rendering Particle System

- Put locations and colors in a buffer and draw as points
- Unfortunately we need to repopulate the VBO each time since the locations changed

```
void ParticlesCPU::draw(){

    glBindVertexArray(VAO);
    glUseProgram(program);

    for(int i=0;i<NUM_PARTICLES;i++){
        particlePoints[i] = particles[i].position;
        particleColors[i] = particles[i].color;
    }

    glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(particlePoints),particlePoints);
    glBufferSubData(GL_ARRAY_BUFFER, sizeof(particlePoints), sizeof(particleColors),particleColors);

    GLuint model_loc = glGetUniformLocation(program,"model_matrix");
    glUniformMatrix4fv(model_loc,1,GL_TRUE,model_matrix);
    GLuint camera_loc = glGetUniformLocation(program,"camera_matrix");
    glUniformMatrix4fv(camera_loc,1,GL_TRUE,camera_matrix);
    GLuint projection_loc = glGetUniformLocation(program,"proj_matrix");
    glUniformMatrix4fv(projection_loc,1,GL_TRUE,projection_matrix);

    glPointSize(3.0);
    glDrawArrays(GL_POINTS,0,NUM_PARTICLES);

}
```

Rendering Particle System

```
void ParticlesCPU::draw(){

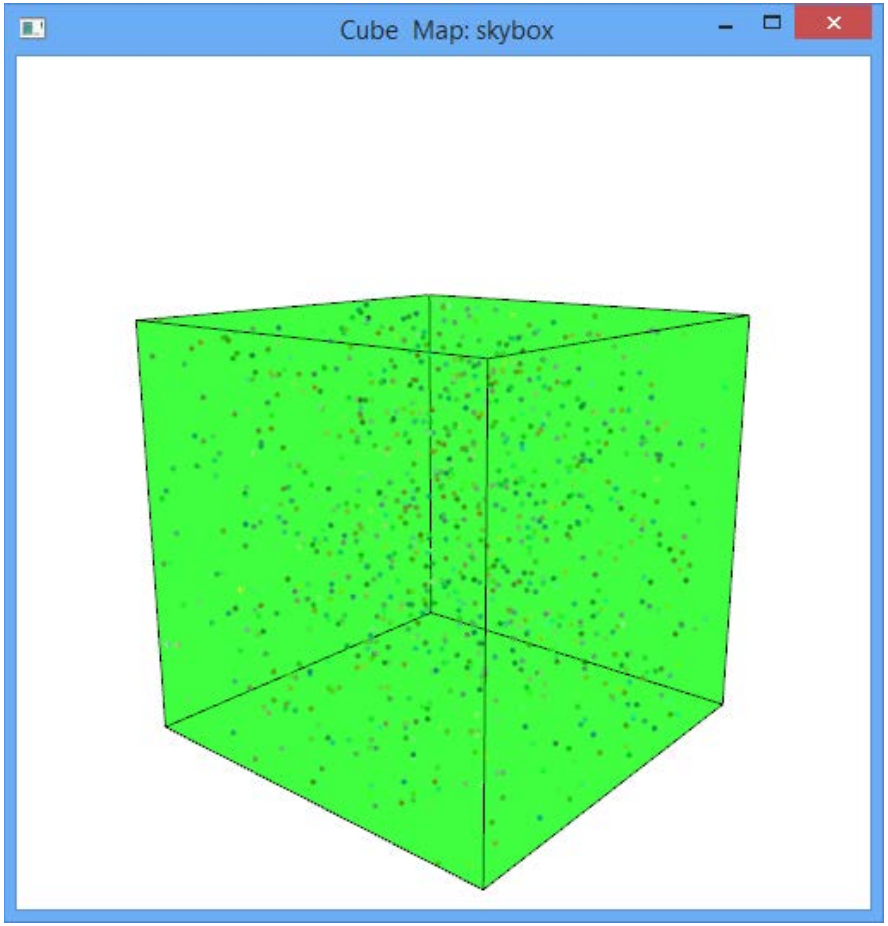
    glBindVertexArray(VAO);
    glUseProgram(program);

    for(int i=0;i<NUM_PARTICLES;i++){
        particlePoints[i] = particles[i].position;
        particleColors[i] = particles[i].color;
    }

    glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(particlePoints),particlePoints);
    glBufferSubData(GL_ARRAY_BUFFER, sizeof(particlePoints), sizeof(particleColors),particleColors);

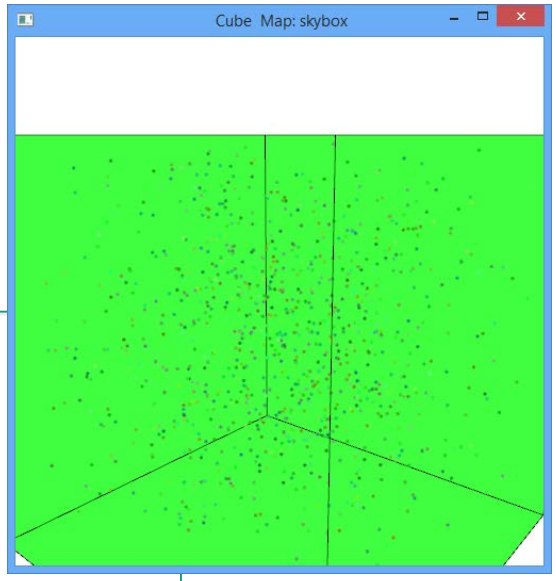
    GLuint model_loc = glGetUniformLocation(program,"model_matrix");
    glUniformMatrix4fv(model_loc,1,GL_TRUE,model_matrix);
    GLuint camera_loc = glGetUniformLocation(program,"camera_matrix");
    glUniformMatrix4fv(camera_loc,1,GL_TRUE,camera_matrix);
    GLuint projection_loc = glGetUniformLocation(program,"proj_matrix");
    glUniformMatrix4fv(projection_loc,1,GL_TRUE,projection_matrix);

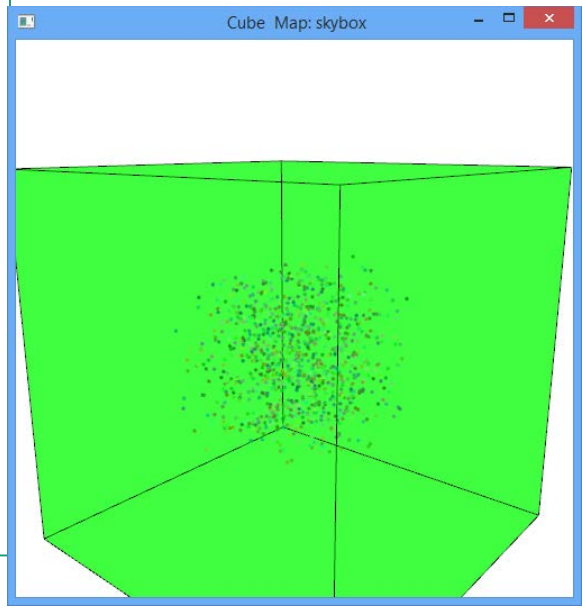
    glPointSize(3.0);
    glDrawArrays(GL_POINTS,0,NUM_PARTICLES);

}
```

Particle System



Flocking

- Change the direction of each particle so it steers towards the center of the system
  - Therefore, each time we must compute the average position
- ```
  vec4 cm;
  for(int k=0; k<3; k++){
       cm[k] = 0;
       for(int i=0; i<num_particles;i++)
            cm[k]+=particles[i].position[k];
       cm[k]/=num_particles;
  }
  ```
- Now we can compute a new velocity that lies between the updated velocity `vector particles[i].velocity` and the vector from the `particles[i].position` to the average position
  ```

Flocking

```
float last_time=0;
void updateParticles(){
    float dt;
    float present_time = glutGet(GLUT_ELAPSED_TIME);
    dt = 0.001*(present_time-last_time);

    float percentFlock=0.01;
    vec4 ap=getAveragePosition();
    for(int i=0; i < NUM_PARTICLES; i++){
        vec4 flockVec = ap-particles[i].position;
        particles[i].velocity = (1.0-percentFlock)*particles[i].velocity+percentFlock*flockVec;
        for(int j=0; j<3; j++){

            particles[i].position[j]+=dt*particles[i].velocity[j];
            particles[i].velocity[j]+=dt*applyForces(i,j)/particles[i].mass;
        }
        testCollision(i);
    }
    last_time = present_time;
}

vec4 getAveragePosition(){
    vec4 cm = vec4(1.0);
    for(int k=0; k < 3; k++){
        cm[k] = 0.0;
        for(int i=0; i<NUM_PARTICLES; i++){
            cm[k]+=particles[i].position[k];
        }
        cm[k]/=NUM_PARTICLES;
    }
    return cm;
}
```





Particles in the GPU

- We said it was a bummer that we need to move data to the GPU each time we want to draw the particle system.
- Can we somehow do more (all?) of the computations in the GPU?
- Just pass initial values in, at each time step update a time variable in the shader
  - Difficult to do collision detection here though ☹

```
void idle(){
    updateParticles();
    glutPostRedisplay();
}

void updateParticles(){
    float elapsed_time = 0.001*glutGet(GLUT_ELAPSED_TIME);

    glUseProgram(programs[passColorShaders]);
    GLuint time_loc = glGetUniformLocation(programs[passColorShaders],"time");
    glUniform1f(time_loc,elapsed_time);


}
```

Particle System (GPU)

```
void initializeParticles(){
    for(int i = 0; i < NUM_PARTICLES; i++){
        particles[i].mass = 1.0;
        for(int j=0; j < 3; j++){
            particles[i].color[j]=(float)rand()/RAND_MAX;
            particles[i].position[j] = 2.0*((float)rand()/RAND_MAX)-1.0;
            particles[i].velocity[j] = 2.0*((float)rand()/RAND_MAX)-1.0;
        }
        particles[i].color.w = 1.0;
        particles[i].position.w = 1.0;
        particles[i].velocity.w = 0.0;
    }

    for(int i=0;i<NUM_PARTICLES;i++){
        particlePoints[i] = particles[i].position;
        particleColors[i] = particles[i].color;
        particleVelocities[i] = particles[i].velocity;
    }

    glBindVertexArray(VAOs[particleVAO]);
    glUseProgram(programs[passColorShaders]);

    glBindBuffer( GL_ARRAY_BUFFER, buffers[particleBuffer]);
    glBufferData( GL_ARRAY_BUFFER, sizeof(particlePoints)+sizeof(particleColors)+sizeof(particleVelocities),NULL, GL_STATIC_DRAW );
    glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(particlePoints),particlePoints);
    glBufferSubData(GL_ARRAY_BUFFER, sizeof(particlePoints), sizeof(particleColors),particleColors);
    glBufferSubData(GL_ARRAY_BUFFER, sizeof(particlePoints)+sizeof(particleColors), sizeof(particleVelocities),particleVelocities);

    vPosition = glGetAttribLocation( programs[passColorShaders], "vPosition" );
    glEnableVertexAttribArray( vPosition );
    glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0) );

    GLuint color_loc = glGetAttribLocation(programs[passColorShaders],"colorIn");
    glEnableVertexAttribArray( color_loc );
    glVertexAttribPointer( color_loc, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(sizeof(particlePoints)));

    GLuint velocity_loc = glGetAttribLocation(programs[passColorShaders],"vVelocity");
    glEnableVertexAttribArray( velocity_loc );
    glVertexAttribPointer( velocity_loc, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(sizeof(particlePoints)+sizeof(particleColors)));
}
```

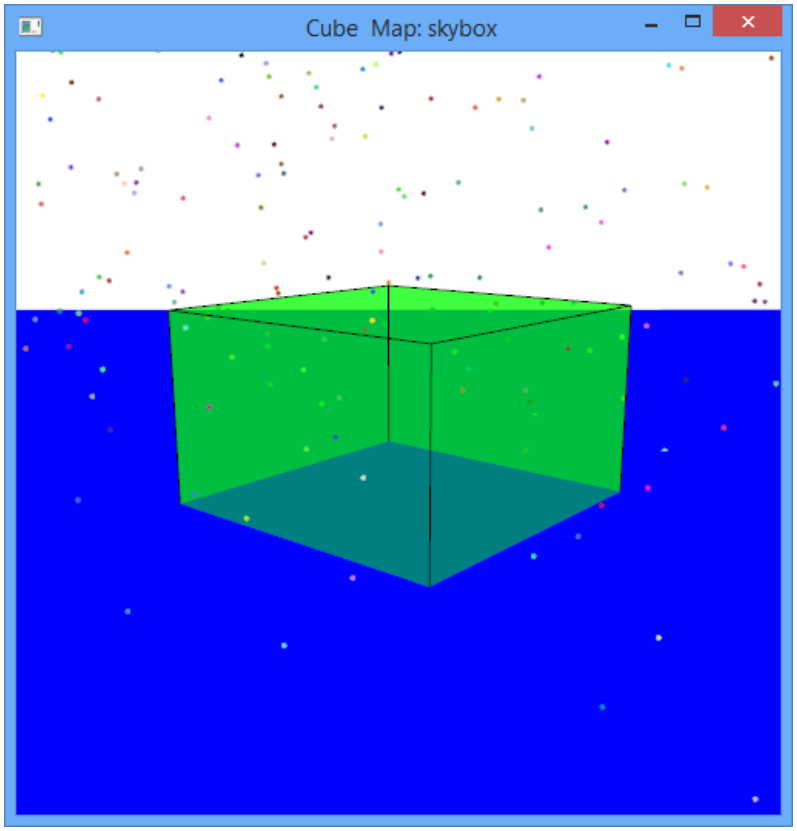
```
void drawParticles(){

    glBindVertexArray(VAOs[particleVAO]);
    glUseProgram(programs[passColorShaders]);

    model_view_loc = glGetUniformLocation(programs[passColorShaders],"model_matrix");
    glUniformMatrix4fv(model_view_loc,1,GL_TRUE,model_matrix);
    camera_view_loc = glGetUniformLocation(programs[passColorShaders],"camera_matrix");
    glUniformMatrix4fv(camera_view_loc,1,GL_TRUE,camera_matrix);
    projection_view_loc = glGetUniformLocation(programs[passColorShaders],"proj_matrix");
    glUniformMatrix4fv(projection_view_loc,1,GL_TRUE,proj_matrix);

    glPointSize(3.0);
    glDrawArrays(GL_POINTS,0,NUM_PARTICLES);

    glBindVertexArray(0);

}
```

Particle System

```
in  vec4 vPosition;
in  vec4 colorIn;
in vec4  vVelocity;

uniform float time;

uniform mat4 model_matrix;
uniform mat4 camera_matrix;
uniform mat4 proj_matrix;

out vec4 color;


void main()
{
        vec3 object_pos;
        object_pos.xyz = vPosition.xyz + vVelocity.xyz*time;

        gl_Position = proj_matrix*camera_matrix*model_matrix*vec4(object_pos,1);
        color = colorIn;


}
```

Wave Motion

- As another example of doing more computations in the shaders let's look at simulating wave motion!



Wave Motion

- Lets create a mesh for the ocean using recursion and quads
- 2 recursions, each generate 4 quads with 4 vertices per quad
  - Total # of vertices = $4*4^2 = 64$

```
void divideQuad(vec4 a, vec4 b, vec4 c, vec4 d, int n){
    if(n==0){
        groundVertices[index] = a; index++;
        groundVertices[index] = b; index++;
        groundVertices[index] = c; index++;
        groundVertices[index] = d; index++;
    }
    else{
        vec4 e = (b+a)/2.0;
        e.w = 1.0;
        vec4 f = (c+d)/2.0;
        f.w = 1.0;
        vec4 g = (d+a)/2.0;
        g.w = 1.0;
        vec4 h = (c+b)/2.0;
        h.w = 1.0;
        vec4 i = (f+e)/2.0;
        i.w = 1.0;

        divideQuad(a,e,i,g,n-1);
        divideQuad(e,b,h,i,n-1);
        divideQuad(g,i,f,d,n-1);
        divideQuad(i,h,c,f,n-1);
    }

}
```

Wave Motion

- Now you can initialize the waves by adjusting the height
- Naïve way is to just assign random heights
  - Easier with indexing
  - Otherwise must set all vertices with the same (x, z) coordinates to the same height, y
- Better?
  - Maybe a Gaussian Mixture Model?

Wave Motion

```
uniform float time;
uniform float xs, zs, // frequencies
uniform float h; // height scale
uniform mat4 ModelView, Projection;
in vec4 vPosition;

void main() {
  vec4 t =vPosition;
  t.y = vPosition.y
     + h*sin(time + xs*vPosition.x)
     + h*sin(time + zs*vPosition.z);
  gl_Position = Projection*ModelView*t;
}
```

```
void updateWaves(){
    float elapsed_time = 0.001*glutGet(GLUT_ELAPSED_TIME);

    glUseProgram(programs[waveShaders]);

    GLuint time_loc = glGetUniformLocation(programs[waveShaders],"time");
    glUniform1f(time_loc,elapsed_time);

    GLuint h = glGetUniformLocation(programs[waveShaders],"h");
    glUniform1f(h,0.2);

    GLuint xs = glGetUniformLocation(programs[waveShaders],"xs");
    glUniform1f(xs,1.0);

    GLuint zs = glGetUniformLocation(programs[waveShaders],"zs");
    glUniform1f(zs,1.0);


}
```

Wave Motion