Particle systems, collision detection, and ray tracing

Computer Graphics
CSE 167
Lecture 17
CSE 167: Computer graphics

- Particle systems
- Collision detection
- Ray tracing
Particle systems

• Used for
  – Fire/sparks
  – Rain/snow
  – Water spray
  – Explosions
  – Galaxies

Based on slides courtesy of Jurgen Schulze
Particle systems

• A particle system is collection of individual elements (particles)
  – Controls a set of particles which act autonomously but share some common attributes
• A particle emitter is a source of new particles
  – 3D point
  – Polygon mesh (particles’ initial velocity vector is normal to surface)
• Particle attributes
  – Position
  – Velocity vector (speed and direction)
  – Color (and opacity)
  – Lifetime
  – Size
  – Shape
  – Weight
Dynamic updates

• Particles change position and/or attributes over time
• Initial particle attributes often created with random numbers

• Frame update
  – Parameters (simulation of particles, can include collisions with geometry)
    • Forces (gravity, wind, etc.) act on a particle
    • Acceleration changes velocity
    • Velocity changes position
  – Rendering
    • GL_POINTS
    • GL_POINT_SPRITE
    • Point shader

Source: http://www.particlesystems.org/
Point rendering, vertex shader

uniform mat4 u_MVPMatrix;
uniform vec3 u_cameraPos;

// Constants (tweakable):
const float minPointScale = 0.1;
const float maxPointScale = 0.7;
const float maxDistance = 100.0;

void main()
{
    // Calculate point scale based on distance from the viewer
    // to compensate for the fact that gl_PointSize is the point
    // size in rasterized points / pixels.
    float cameraDist = distance(a_position_size.xyz, u_cameraPos);
    float pointScale = 1.0 - (cameraDist / maxDistance);
    pointScale = max(pointScale, minPointScale);
    pointScale = min(pointScale, maxPointScale);

    // Set GL globals and forward the color:
    gl_Position  = u_MVPMatrix * vec4(a_position_size.xyz, 1.0);
    gl_PointSize = a_position_size.w * pointScale;
    v_color      = a_color;
}
Particle systems

• Demo in WebGL

https://nullprogram.com/webgl-particles/
References

• Tutorial with source code by Bartlomiej Filipek, 2014
  https://www.codeproject.com/Articles/795065/Flexible-particle-system-OpenGL-Renderer

• Articles with source code

• Founding scientific paper:
Collison detection
Collision detection

• Goals
  – Physically correct simulation of collision of objects
    • Not covered in this course
  – Determine if two objects intersect

• Slow calculation because of exponential growth $O(n^2)$
  – Number of collision tests $n(n - 1) / 2$
Intersection test

• Purpose
  – Keep moving objects on the ground
  – Keep moving objects from going through walls, each other, etc.

• Goal
  – Believable system, does not have to be physically correct

• Priority
  – Computationally inexpensive

• Typical approach
  – Spatial partitioning
  – Object simplified for collision detection by one or a few
    • Points
    • Spheres
    • Axis aligned bounding box (AABB)
  – Pairwise checks between points/spheres/AABBs and static geometry
Sweep and prune algorithm

• Sorts bounding boxes
• Not intuitively obvious how to sort bounding boxes in 3D
• Dimensionality reduction approach
  – Project each 3D bounding box (cuboid) onto the X, Y, and Z axes
  – Find overlaps in 1D
    • A pair of bounding boxes overlaps if and only if their intervals overlap in all three dimensions
      – Construct 3 lists, one for each dimension
      – Each list contains start/end point of intervals corresponding to that dimension
      – By sorting these lists, we can determine which intervals overlap
      – Reduce sorting time by keeping sorted lists from previous frame, changing only the interval endpoints
Collision map (CM)

- 2D map with information about where objects can go and what happens when they go there
- Colors indicate different types of locations
- Map can be computed from 3D model (or hand drawn)
- Granularity defines how much area (in object space) one CM pixel represents
Ray tracing
Projection

• To render an image of a scene, we project the 3D scene to the 2D image plane
• Most common projection type is perspective projection

Based on slides courtesy of Steve Marschner
Two approaches to rendering

for each object in the scene {
    for each pixel in the image {
        if (object affects pixel) {
            do something
        }
    }
}

object order
    or
rasterization

for each pixel in the image {
    for each object in the scene {
        if (object affects pixel) {
            do something
        }
    }
}

image order
    or
ray tracing
Ray tracing idea

- Start with a pixel—what belongs at that pixel?
- Set of points that project to a point in the image: a ray
Ray tracing idea

viewer (eye) → viewing ray → visible point → objects in scene

light source → illumination
Ray tracing algorithm

for each pixel {
    compute viewing ray
    intersect ray with scene
    compute illumination at visible point
    put result into image
}
Generating eye rays, orthographic projection

- Ray origin (varying): pixel position on viewing window
- Ray direction (constant): view direction
Generating eye rays, perspective projection

- Ray origin (constant): viewpoint
- Ray direction (varying): toward pixel position on viewing window
Software interface for cameras

• Key operation: generate ray for image position

```java
class Camera {
    ...
    Ray generateRay(int col, int row);
}
```

• Modularity problem: camera should not have to worry about image resolution
  – Better solution: normalized coordinates

```java
class Camera {
    ...
    Ray generateRay(float u, float v);    // args go from 0, 0 to 1, 1
}
```
Specifying views in Ray 1

<camera type="PerspectiveCamera">
    <viewPoint>10 4.2 6</viewPoint>
    <viewDir>-5 -2.1 -3</viewDir>
    <viewUp>0 1 0</viewUp>
    <projDistance>6</projDistance>
    <viewWidth>4</viewWidth>
    <viewHeight>2.25</viewHeight>
</camera>

<camera type="PerspectiveCamera">
    <viewPoint>10 4.2 6</viewPoint>
    <viewDir>-5 -2.1 -3</viewDir>
    <viewUp>0 1 0</viewUp>
    <projDistance>3</projDistance>
    <viewWidth>4</viewWidth>
    <viewHeight>2.25</viewHeight>
</camera>
Pixel-to-image mapping

• Mapping to normalized coordinates

\[
\begin{align*}
    u &= (i + 0.5)/n_x \\
    v &= (j + 0.5)/n_y
\end{align*}
\]
Ray intersection
Ray: a half line

- Standard representation: point \( \mathbf{p} \) and direction \( \mathbf{d} \)
  \[ r(t) = \mathbf{p} + t\mathbf{d} \]
  - This is a parametric equation for the line
  - Lets us directly generate the points on the line
  - If we restrict to \( t > 0 \) then we have a ray
  - Note replacing \( \mathbf{d} \) with \( \alpha \mathbf{d} \) does not change ray (\( \alpha > 0 \))
Ray-sphere intersection: algebraic

• Condition 1: point is on ray
  \[ r(t) = p + td \]

• Condition 2: point is on sphere
  – Assume unit sphere
  \[ \|x\| = 1 \iff \|x\|^2 = 1 \]
  \[ f(x) = x \cdot x - 1 = 0 \]

• Substitute:
  \[ (p + td) \cdot (p + td) - 1 = 0 \]
  – This is a quadratic equation in \( t \)
Ray-sphere intersection: algebraic

• Solution for $t$ by quadratic formula

$$
t = \frac{-d \cdot p \pm \sqrt{(d \cdot p)^2 - (d \cdot d)(p \cdot p - 1)}}{d \cdot d}
$$

$$
t = -d \cdot p \pm \sqrt{(d \cdot p)^2 - p \cdot p + 1}
$$

– Simpler form holds when $d$ is a unit vector but we will not assume this in practice

– Unit vector form is used to make the geometric interpretation (next slide)
Ray-sphere intersection: geometric

\[ t_m = -p \cdot d \]
\[ l_m^2 = p \cdot p - (p \cdot d)^2 \]
\[ \Delta t = \sqrt{1 - l_m^2} \]
\[ = \sqrt{(p \cdot d)^2 - p \cdot p + 1} \]
\[ t_{0,1} = t_m \pm \Delta t = -p \cdot d \pm \sqrt{(p \cdot d)^2 - p \cdot p + 1} \]
Ray-triangle intersection

• Condition 1: point is on ray
  \[ r(t) = p + td \]

• Condition 2: point is on plane
  \[(x - a) \cdot n = 0\]

• Condition 3: point is on the inside of all three edges

• First solve 1 and 2 (ray–plane intersection)
  – Substitute and solve for \( t \)
    \[(p + td - a) \cdot n = 0\]
    \[t = \frac{(a - p) \cdot n}{d \cdot n}\]
Ray-triangle intersection

- In plane, triangle is the intersection of 3 half spaces
Deciding about insideness

• Need to check whether hit point is inside 3 edges
  – Easiest to do in 2D coordinates on the plane
• Will also need to know where we are in the triangle
  – For textures, shading, etc.
• Efficient solution
  – Transform to coordinates aligned to the triangle
Barycentric coordinates

- A coordinate system for triangles
  - Algebraic viewpoint
    \[ \mathbf{p} = \alpha \mathbf{a} + \beta \mathbf{b} + \gamma \mathbf{c} \]
    \[ \alpha + \beta + \gamma = 1 \]
  - Geometric viewpoint (areas)
- Triangle interior test:
  \[ \alpha > 0; \quad \beta > 0; \quad \gamma > 0 \]

[Shirley 2000]
Barycentric coordinates

• A coordinate system for triangles
  – Geometric viewpoint (distances)
  – Linear viewpoint (basis of edges)

\[ \alpha = 1 - \beta - \gamma \]
\[ p = a + \beta(b - a) + \gamma(c - a) \]
Barycentric coordinates

- Linear viewpoint (basis for the plane)

- In this view, the triangle interior test is

\[ \beta > 0; \quad \gamma > 0; \quad \beta + \gamma < 1 \]
Barycentric ray-triangle intersection

• Every point on the plane can be written in the form:
  \[ p = a + \beta(b - a) + \gamma(c - a) \]
  for some numbers \( \beta \) and \( \gamma \)

• If the point is also on the ray then it is
  \[ p + td \]
  for some number \( t \)

• Set them equal: 3 linear equations in 3 variables
  \[ p + td = a + \beta(b - a) + \gamma(c - a) \]
  then solve them to get \( t \), \( \beta \), and \( \gamma \)
Barycentric ray-triangle intersection

\[ p + td = a + \beta(b - a) + \gamma(c - a) \]
\[ \beta(a - b) + \gamma(a - c) + td = a - p \]

\[
\begin{bmatrix}
    a - b & a - c & d
\end{bmatrix}
\begin{bmatrix}
    \beta \\
    \gamma \\
    t
\end{bmatrix} = \begin{bmatrix}
    a - p
\end{bmatrix}
\]

\[
\begin{bmatrix}
    x_a - x_b & x_a - x_c & x_d \\
    y_a - y_b & y_a - y_c & y_d \\
    z_a - z_b & z_a - z_c & z_d
\end{bmatrix}
\begin{bmatrix}
    \beta \\
    \gamma \\
    t
\end{bmatrix} = \begin{bmatrix}
    x_a - x_p \\
    y_a - y_p \\
    z_a - z_p
\end{bmatrix}
\]

Cramer’s rule is a good fast way to solve this system
Ray intersection in software

- All surfaces need to be able to intersect rays with themselves

```java
class Surface {
    ...
    abstract boolean intersect(IntersectionRecord result, Ray r);
}
```

```java
class IntersectionRecord {
    float t;
    Vector3 hitLocation;
    Vector3 normal;
    ...
}
```
Image so far

- With eye ray generation and sphere intersection

```java
Surface s = new Sphere((0.0, 0.0, 0.0), 1.0);
for 0 <= iy < ny
    for 0 <= ix < nx {
        ray = camera.getRay(ix, iy);
        hitSurface, t = s.intersect(ray, 0, +inf)
        if hitSurface is not null
            image.set(ix, iy, white);
    }
```
Ray intersection in software

• Scenes usually have many objects
• Need to find the first intersection along the ray
  – That is, the one with the smallest positive $t$ value
• Loop over objects
  – Ignore those that do not intersect
  – Keep track of the closest seen so far
  – Convenient to give rays an ending $t$ value for this purpose (then they are really segments)
Intersection against many shapes

- The basic idea

```java
intersect (ray, tMin, tMax) {
    tBest = +inf; firstSurface = null;
    for surface in surfaceList {
        hitSurface, t = surface.intersect(ray, tMin, tBest);
        if hitSurface is not null {
            tBest = t;
            firstSurface = hitSurface;
        }
    }
    return hitSurface, tBest;
}
```

- This is linear in the number of shapes but there are sublinear methods (acceleration structures)
Summary of CSE 167

• Geometric transformations
• Coordinate frames
• Projection and viewing
• Rasterization
• Surface shading: materials and lights
• Graphics pipeline
• Triangle meshes
• Texture mapping
• Scene graph
• Curves

• Culling
• Environment mapping
• Toon shading
• Surface patches
• Procedural modeling
• Shadow mapping
• Shadow volumes
• Deferred rendering
• Particle systems
• Collision detection
• Ray tracing