Virtual Environments
Jean-Christian Delannoy

- viewing in 3D
- world coordinates / geometric modeling
- culling
- lighting
- virtualized reality
- collision detection
- collision response
- interactive forces / object interaction
Viewing in 3D
Viewing in 3D
Families of Perspective and Parallel Projections

- **PERSPECTIVE**
  - converging projection rays / projectors

- **PARALLEL**
  - parallel projectors

- **OBLIQUE**
  - projectors are not perpendicular to the projection plane

- **ORTHOGRAPHIC**
  - projectors are perpendicular to the projection plane

- **MULTIVIEW**
  - projection plane is parallel to the principal planes

- **AXONOMETRIC**
  - projection plane is parallel to the principal planes
Parallel Projection

Parallel projection from a 3D space onto a projection plane

Accomplished in 2 steps:

• Transform the projection plane to match the $xy$ plane of the 3D space

• Remove the $z$ component from all viewable points
Parallel -> Orthographic -> Multi-view

- Each projection plane is perpendicular to a principal axis (x, y or z)
- Distances and angles of the object can be measured in these projections but each image only depicts one face of a 3D object.
- Heavily used in 3D modeling and in engineering for computer-aided design.
Perspective Projection

- Determines object sizes based on the object’s distance from the projection plane.
- With this projection, the “light rays” bouncing off the object in the 3D space converge on a single point in the viewer’s eye.
**Perspective Projection**

Specifying an arbitrary view

- **VPN** (view plane normal)
- **VUN** (view up normal)
- **VRP** (view reference point)

2D screen

- Extract
  - Translation matrix
  - Rotation matrix
  - Scaling matrix

- Combine
  - Final viewing matrix

- VertexA *= final viewing matrix
  - VertexB *= final viewing matrix
  - VertexH *= final viewing matrix
Perspective Projection
the frustum

- The frustum is the volume of the 3D world which will be displayed on the 2D screen.

- It is defined by three properties
  - far plane
  - near plane
  - field of view
Clipping & Culling
Clipping \ Culling

Rendering polygons on screen takes up lots of processing power therefore means must be taken of only rendering what is going to be seen.

Culling is the act of not rendering polygons or objects which will not be seen on the screen.

- **frustum culling**: not drawing objects which are not in the frustum.

- **backface culling**: not drawing polygons which are not facing the camera.

- **ray-traced culling**: not drawing polygons which are behind other objects.
A simple algorithm for frustum culling:
\set-up 6 planes with frustum sides and normals heading outward
bool bDisplay = true
for (int i=0; i<6; ++i)
    if (DistanceFromPlane( plane(i), vPosition ) > fRadius)
    { bDisplay = false; }

Frustum Culling
Frustum Culling continued

Frustum culling is easy for dynamic (stand-alone) objects but is more complicated for the much larger static environment.

- Cannot display all polygons.
- Checking each poly to see if in frustum is not efficient enough.
- Solutions later on ……
Polygons

reminder

A polygon is a triangle with 3 vertices in the 3D world and one normal

Two ways to specify normals (examples in OpenGL)

• manually

```c
glBegin(GL_TRIANGLE)
    glNormal3f(0, 1, 0)
    glVertex3f(0.5, 0.0, 0.5)
    glVertex3f(-0.5, 0.0, 0.5)
    glVertex3f(0.5, 0.0, -0.5)
glEnd()
```

• polygon winding: normals are generated based on the order vertices are specified

```c
glEnable(GL_NORMALIZE)
glBegin(GL_TRIANGLE)
    glVertex3f(0.5, 0.0, 0.5)
    glVertex3f(-0.5, 0.0, 0.5)
    glVertex3f(0.5, 0.0, -0.5)
glEnd()
```
Backface Culling

Once again, rendering is expensive therefore only polygons which are facing the camera will be displayed. Hence, backface culling

\[ \text{VPN} \cdot N < 0 \]
therefore draw polygon

\[ \text{VPN} \cdot N > 0 \]
therefore do NOT draw polygon
Light Modeling
Light Modeling

types of lighting

Ambient lighting

Ambient light is the type of light which fills a room and does not come from any particular direction. All sides of the 3D model are lit at the same intensity.

Diffuse lighting

The type of light which comes from a particular direction and is reflected evenly across a surface. Two surfaces at different angles will reflect the light differently. (i.e.: sunlight)

Specular lighting

Like the diffuse light except the light originates from a single point.
**Diffuse Light**

flat shading

* n.b. We can define a light intensity for every vertex on every polygon

\[
\text{light intensity} = - (\mathbf{L} \cdot \mathbf{N})
\]

- if \( \mathbf{N} \neq -\mathbf{L} \), light intensity is 0.0
- if \( \mathbf{N} == -\mathbf{L} \), light intensity is 1.0

- Vertices on a polygon will all have the same light intensity
Diffuse Light
smooth shading

We define a **vertex normal** as being equal to the average of the normals of all polygons which it is part of.

\[
\text{light intensity} = - (\mathbf{L} \cdot \mathbf{N})
\]

- Vertices on a polygon will **NOT** all have the same light intensity.
- Lighting is more progressive and has a more natural feel.
- Vertices on a polygon will **NOT** all have the same light intensity.
Virtualized Reality
Virtualized Reality basics

When the virtual world is based on sensor information about the real world objects and phenomena it becomes virtualized reality.

How we can accomplish this.

```
Class Square {
    public: float mass = 1000;
    ...
}
```

REAL WORLD  VIRTUAL WORLD
A Few Object Properties

Position (vector)
Represents the object’s position in the 3D environment.

Orientation (3 orthogonal vectors)
Triad of vectors which represent the object’s orientation in the 3D environment.

Mass (floating point)
A force applied to an object with a higher moment of inertia will have less effect than if the same force is applied to an object with a lower mass.

Moment of Inertia (floating point)
A torque applied to an object with a higher moment of inertia will have less effect than the same torque applied to an object of lower moment of restitution.

Restitution coefficient (floating point)
During a collision, the restitution coefficient of the involved objects will determine the elasticity of the collision. Solid objects typically have a higher restitution than soft objects.
A Few Object Properties
(continued)

Linear drag coefficient (floating point)
Determines the magnitude of the air friction on the object’s front axis.

Orthogonal drag coefficient (floating point)
Determines the magnitude of the air friction on the object’s right and top axes.

Velocity (vector)
Represents the object’s velocity in the 3D world.

Angular velocity (vector)
Represents the object’s angular velocity in the 3D world.

\[
\begin{align*}
v_{\text{Top}} & \quad \text{x: rotation speed around } v_{\text{Right}} \text{ axis.} \\
v_{\text{Front}} & \quad \text{y: rotation speed around } v_{\text{Top}} \text{ axis.} \\
v_{\text{Right}} & \quad \text{z: rotation speed around } v_{\text{Front}} \text{ axis.}
\end{align*}
\]
**Physics Engines**

**Why? What? Where? How?**

- Many virtual reality applications only account for 3D geometry and lighting of objects.
- A physics engine will take care of all the dynamic movements and interactions of objects in the virtual environment. These movements will be based on the physical properties of the objects and basic mechanic properties of our world.
- Accurate physics modeling can be used in many applications
  - virtual prototyping environments.
  - Real-world simulators for engineering design.
  - More immersive video games and virtual reality applications.
- This project dealt with the incorporation of the mechanical modeling of collision detection and response, interactive forces and friction.
- Was built using OpenGL and C++ to be used in video game applications.
Collision Detection and Response
Collision Detection with Spheres

if( |pos_1 - pos_2| < radius_1 + radius_2 )
    return true; // collision
else return false;

Collision Normal

vCollisionNormal = pos_1 - pos_2;
Linear Collision Response

- Orthogonal velocities of objects are not affected.
- Only the components of the velocities which are along the collision normal will be affected. \((v_{\text{NormalVelocity}} = (v_{\text{Velocity}} \cdot v_{\text{Normal}}) \ast v_{\text{Normal}})\)
- The post-collision velocities are determined thanks to the **impulse method**.
- The impulse method combines the physical equations of conservation of momentum and conservation of energy.

\[
\begin{align*}
\sum_{i=1}^{m} m_i v_i^2 &= \sum_{i=1}^{m} m_i v_i^2 \\
J &= \left( v_1 \cdot n \right) \cdot \left( v_2 \cdot n \right) \cdot (e \cdot 1) \\
\frac{1}{m_1} &= \frac{1}{m_2} \\
\frac{v_1}{m_1} &= \frac{J \cdot n}{m_1} \\
\frac{v_2}{m_2} &= \frac{J \cdot n}{m_2}
\end{align*}
\]

\(e\): restitution coefficient
\(n\): collision normal
\(m\): object mass
\(v\): object velocity
OBBs
oriented bounding boxes

• Instead of performing collision detection on each and every polygon of our model we will define a much simpler *outer shell* which will be its volume.
Collision Detection with OBBs

bool bCollision = true;
for(i = 0; i < verticesA.size; ++i)
    for(j = 0; j < planesB.size; ++j)
        if(planesB[j].distance(verticesA[i]) > 0)
            bCollision = false; // no collision
return bCollision;

Collision Normal

vCollisionNormal =
closestPlane.Normal();
Linear and Angular Collision Response
modifies linear and angular velocities

- The only difference is that the momentum equation is now:

\[ m v_{total} \neq m v_{linear} \neq I(\omega \neq r) \]

- The same characteristic equations are used for the linear and angular collision response routines as the ones used for linear collision response. The impulse method becomes the following:

\[
J \neq \frac{1}{m_1} \left( \frac{m_2}{m_2} \left( \frac{I_1}{r_1 \cdot n} \right) \left( \frac{r_1}{n} \right) \right) \neq \frac{1}{m_2} \left( \frac{I_2}{r_2 \cdot n} \right) \neq \frac{1}{I_1} \left( \frac{I_1}{r_1 \cdot n} \right) \neq \frac{1}{I_2} \left( \frac{I_2}{r_2 \cdot n} \right)
\]

where \( v_t = v_{linear} + (\omega \neq r) \)
Interactive Forces
**Decomposition of Applied Forces**

- All the forces which are applied in the virtual environment affect bodies in accordance with the second laws of Newton regarding mass and moment of inertia.

\[ F = ma \quad \text{and} \quad M = I \cdot \omega \]

- Both the linear and angular velocities will be modified whenever a force is applied on a line which does not intersect the center of mass since torque will be generated. Forces applied to an object are therefore first decomposed into the component which is applied directly through the center of mass and the component which causes torque.

- The component through the center of mass will modify the linear acceleration while the remainder will modify the angular acceleration.
Implementation

Object.vTotalForce = (0, 0, 0);
Object.vTotalTorque = (0, 0, 0);

Object.ApplyForces();

\[
\begin{align*}
\text{vTotalForce} & = \text{vForce} \cdot \text{vR} \\
\text{vTotalTorque} & = \text{vForce} \cdot \text{vR} \\
\text{vAcceleration} & = \frac{\text{vTotalForce}}{\text{Object.fMass}} \\
\text{vAngularAcceleration} & = \frac{\text{vTotalTorque}}{\text{Object.fInertia}} \\
\text{Object.vVelocity} & += \text{vAcceleration} \cdot \text{fTimeElapsed} \\
\text{Object.vAngularVelocity} & += \text{vAngularAcceleration} \cdot \text{fTimeElapsed} \\
\text{Object.vPosition} & += \text{Object.vVelocity} \cdot \text{fTimeElapsed} \\
\text{Object.vOrientation} & += \text{Object.vAngularVelocity} \cdot \text{fTimeElapsed}
\end{align*}
\]
Applying Forces for Object Control

- By simply enabling and disabling certain forces (thrusters) on the spaceship we achieve natural motion in any direction.

Thruster topology on a 3D rigid body

\[
\begin{align*}
  a &: \text{ enable top right thruster} \\
  s &: \text{ enable bottom right thruster} \\
  d &: \text{ enable top left thruster} \\
  f &: \text{ enable bottom left thruster} \\
  \text{space} &: \text{ enable main thruster}
\end{align*}
\]
Questions?