Graphics Concepts

3D Topics

Rendering  graphics pipeline, render loop,
Modeling  geometries, materials, textures, lights, modelers
Interacting select, manipulate

3D graphics algorithms / programming concepts

collections  geometry, transforms
Math  matrix composition, dot and cross products
interaction  selection: pick rays, high lighting
animation  motion, collision
behaviors  AI – path finding, flocking, chase / evade, NPC
modeling  procedural, 3D modelers
From model to view

World object db (world coordinates)
\[\downarrow\]
Object transforms (wc)
\[\downarrow\]
Cull / clip non-viewable objects (vc)
\[\downarrow\]
Viewing transform (view coordinates)
\[\downarrow\]
Render (lights, surface properties) (vc)
Graphics Pipeline

Draw a frame

model space

→

vertex effect (shader)

→

pixel effect (shader)

Z buffer

→

frame buffer

→

monitor

Application space

world & view transformations, lighting, projection transformations

transformed points: screen position, color (lighting), normals.

pixel color and depth

GPU

Display device
Render Loop

Net based render loop

Create (object & action db)

Read sensor values

Process time dependent actions

Receive & apply network events

Render world (graphics pipeline)

Frame rate

Network events

Shared virtual environment server
MonoGames

MonoGames is a framework for developing Managed C# games for most desktop and mobile systems. MonoGames projects will create:

- Program.cs – class creates a Game1 instance and invokes its run()
  ```csharp
  static void Main(string[] args) {
      using (Game1 game = new Game1()) { game.Run(); } }
  ```

- Game1.cs – class that derives (extends) Game
  ```csharp
  public class Game1 : Microsoft.Xna.Framework.Game {
  ```

- Content subdirectory – contains *.x model and texture files and other "content resources"

  ```csharp
  public class Game1 : Microsoft.Xna.Framework.Game {  
    public Game1() {...}  
    protected override void Initialize {...}  
    protected override void LoadContent() {...}  
    protected override void UnloadContent() {...}  
    protected override void Update(GameTime gameTime) {...}  
    protected override void Draw(GameTime gameTime) {...}  
  }
  ```
**Game class**

The *GameName* class derives from Game.

**Game initialization sequence**

- **Constructor** – creates a GraphicsDeviceManager
  - sets Content directory

  ```csharp
graphics = new GraphicsDeviceManager(this);
Content.RootDirectory = "Content";
```

- **Initialize method** – called after construction but before `LoadContent`. Developer can initialize and GameComponents they added.

- **LoadContent method** – add "content" files
  ```
  { models *.x, textures, effects *.fx, ... }
  ```
Game loop: Update(), Draw()

Game hides the rendering loop from the developer.

- **IsFixedTimeStep** – property determines interval of calls to Update method – contains changes in "logic" of the game: rotation changes, user input (keyboard, gamepad)

```csharp
protected virtual void Update (GameTime gameTime)
```

*gameTime* is the elapsed time of program execution

Draw method – contains drawing requests for meshes, sprites, and primitives. Called as frequently as possible.

```csharp
protected virtual void Draw (GameTime gameTime)
```
Reference Axes

3D space (with 6 DFs) is usually described with a reference axis.

MonoGames uses a right-hand coordinate system
positive Z towards viewer

Positive rotations about an axis are counter-clockwise looking along the axis to the origin.

Or, clockwise looking from the origin along the axis.

Place right hand around "axis" with thumb pointing along the positive direction and the finger curl is in the positive direction.
Object in 3 space

Consider an object in 3 space. It has a:

- **position** ( @ )  point, object's center ("anchor") location
- **orientation**  "local reference frame"
  or axes: **up, right, backward**

Matrix.Identity represents a 3D reference axes with homogeneous coordinates:
(10,-5,0,1) a point and (10,-5,0,0) a vector)

A matrix (Orientation Matrix) can also represent an object's local reference frame.  \[
\begin{bmatrix}
X & Y & Z & H
\end{bmatrix}
\]

- **Right** \[
\begin{bmatrix}
1 & 0 & 0 & 0
\end{bmatrix}
\]
- **Up** \[
\begin{bmatrix}
0 & 1 & 0 & 0
\end{bmatrix}
\]
- **Backward** \[
\begin{bmatrix}
0 & 0 & 1 & 0
\end{bmatrix}
\]
- **Translation** \[
\begin{bmatrix}
0 & 0 & 0 & 1
\end{bmatrix}
\]

Rows are the local reference frame vectors and center point.
Columns are their coordinates on the reference axes.
World View Projection

MonoGames has separate matrices for transformations of:

World  objects specified in world coordinates (reference axes)
Update(Gametime gt) method sets up transformations.

View  objects transformed to view coordinates – the camera's reference frame becomes the reference axes.
Often set in LoadContent()

Projection  objects transformed to 2D display plane coordinates (3D clipping).
Often set in LoadContent()

World, View, and Projection transforms are applied in Draw(Gametime gt).
BasicEffect

BasicEffect (subclasses Effect) represents shader model 1.1 effects

Matrix worldMatrix, viewMatrix, projectionMatrix;
BasicEffect be;
...
be = new BasicEffect(graphics.GraphicsDevice, null);
...
be.Alpha = 1.0f;
be.DiffuseColor = new Vector3(1.0f, 0.0f, 1.0f);
be.SpecularColor = new Vector3(0.25f, 0.25f, 0.25f);
be.SpecularPower = 5.0f;
be.AmbientLightColor = new Vector3(0.75f, 0.75f, 0.75f);
...
be.LightingEnabled = true;

BasicEffect has 3 DirectionalLights: 0, 1, 2
3 matrices: world, view, projection
Set Camera, draw

Matrix.CreateLookAt(
    Vector3 cameraPosition,
    Vector3 lookAtPosition,
    Vector3 cameraUpVector)

Matrix.CreateLookAt(
    new Vector3(20, 50, 50),
    Vector3.Zero, Vector3.Up);
Vertex based "shapes"

Models (*.x) loaded contain vertices and indices.

Shapes can be defined/drawn by surface.

This pyramid has 6 surfaces:
- 4 on the sides
- 2 on the bottom

These surfaces are defined by 5 vertices ("points")
- Each vertex has a position – a Vector3 (x, y, z)
- Vertices can also have a color or a texture and a normal

Vertices are placed in a VertexBuffer and can be stored on the GPU

These surfaces can be drawn with the Primitive Type TriangleList.
- duplicate copies of vertices in the VertexBuffer
- TriangleList would have 3 vertex values.
- 18 vertices = 3 / surface * 6 surfaces
Vertex Buffer, Indices & IndexBuffer

// create VertexBuffer
vb = new VertexBuffer(display,
typeof(VertexPositionColor),
    vertexColored.Length, BufferUsage.WriteOnly);

// and store on GPU
vb.SetData<VertexPositionColor>(vertex, 0,
    vertex.Length);

Duplicate vertices can be eliminated from the VertexBuffer by having an IndexBuffer reference VertexBuffer vertices.

private IndexBuffer ib = null;
// define face vertex index clockwise from camera view
private int[][] indices = {...};
...

// create IndexBuffer and store on GPU
ib = new IndexBuffer(display,typeof(int), indices.Length,
    BufferUsage.WriteOnly);
IB.SetData<int>(indices);
Normals

Normals are vectors orthogonal to a plane. The cross product of 2 vectors sharing a vertex is its normal.

\[
\text{Vector3.Cross}(\text{vector1}, \text{vector2});
\]

MonoGames has a VertexPositionNormalTexture format.

\[
\text{VertexPositionNormalTexture vertexPNT} =
\text{new VertexPositionNormalTexture(}
\text{new Vector3}(x, y, z), \quad \text{// position}
\text{Vector3.Up,} \quad \quad \quad \quad \text{// normal "up" (0, 1, 0)}
\text{new Vector2}(u, v)); \quad \text{// texture coordinate}
\]

Normals used for:
- surface visibility
- surface shading (surface lighting reflection)

A normalized vector has a length of 1
- a unit vector.

\[
\text{Vector3.Normalize(aVector3)};
\]
Textures

Textures are 2D images that can be used as a surface’s “material”.

Texture coordinates are \((u,v)\) and range from \(0.0 \ldots 1.0\)

A triangle with vertices having \((0,0), (0, 1), (1,1)\) \(u,v\) coordinates and the DirectX puck.bmp texture.

MonoGames Texture2D

Texture2D puckTexture = Content.Load<Texture2D>("puck");
Modeling Landscape

Landscape – a grid of vertices with height displacement.

Height map generation:

Modeler – user interaction (*presentation*)
- complete control
- time intensive / run
- connect vertices to animation

Algorithm – parameterized
- random control (tweak)
- quick / run
- randomizing functions: fractals, oscillators, …
- connect vertices to animation

hybrid – "paint" or texture parameters, *variable control
- time / run
- pipeline processing of parameters
Consider a movable 6DF object as a "Rigid body" with rotation and translations forces that affect its movement position ( @ ) point, object's center ("anchor") location target (vector in forward direction) length is distance of move rotation – sets "local" or "frame" axis: up, right, backward),

Steering forces:
rotation radians about an (arbitrary) axis
translation vector (length = magnitude of force)

0 Create an object-to-origin translation (OT)
(a - @ translation, also create its inverse OT⁻¹)
1 Concatenate rotational forces into RotationMatrix (RM)
2 Sum translations forces (concatenate if using Matrix methods) into TranslationMatrix (TM)
3 \( OM = OM \times OT \times RM \times OT^{-1} \times TM \)
Tours: camera animation

Consider a path (an array of Matrix where each Matrix is a position and orientation (orientation matrix) of "nodes" in the linear "graph".

As a function of time Update(GameTime) can place the current camera at the next "node" in the path.

To smooth animation between nodes intermediate "nodes" can be calculated using Matrix.Lerp(...)

Lerp – linear interpolants between two matrices based on an amount (0..1).

```
public static Matrix Lerp ( Matrix matrix1, Matrix matrix2, float amount )

public static Vector3 Lerp (Vector3 vector1, Vector3 vector2, float amount)
```
Collision Detection: computationally intense

2 or more objects are moving - do they collide?

$O(n^2) \quad n = \text{faces} / \text{object}$

How often is collision tested (collision culling)

Test when predictive collision is possible

Multi-phase approach (simple example)

Cull all objects pairs that exceed collision thresholds (distance from centers)

Test spherical bounding volumes

Test axis - aligned box bounding volumes

Test axis - oriented box bounding volumes

Test each surface for collision

Spherical test \quad (r_1 + r_2 \geq \text{distance between centers})

often sufficient.
Obstacle Avoidance, Flocking

Obstacles avoidance requires "see ahead" information. Line of sight w/ Ray.intersects(BoundingSphere)

Agents see or have "sensors" ahead of them.

Sensors detect potential collisions with obstacles. Detection determines avoidance steering

C. Reynold’s flocking algorithm’s 3 rules:

Cohesion – boid steers toward the average position of its neighbors

Alignment – boid aligns to average alighment of its neighbors

Separation – boids steer to avoid hitting neighbors

Boids have a unit of visibility: arc radius, r, and an angle $\theta$
Navigation graphs

Information about the local area can be stored at locations in the scene: navigation nodes, way points, path (nodes), ...

"cost" to traverse terrain  // distance to source + est. distance to goal
objects in local area

Nodes are in a connected graph.

Nodes provide a limited visibility detection – can detect n nodes from current node.

Nodes limit navigation along "correct" paths.

Movable objects translation is interpolated along path.

Nodes can be placed by designer, equally spaced from a terrain grid, by space partitioning (Quad or Oct Tree).
Game Intelligence: Controllers

Expert System – brute force controller "knows all" about the "game"

- size of KB (knowledge base)

Searching controllers: explicit goals -- search for action

Reactive controllers: State $\rightarrow$ Action

- as game becomes complex state determination becomes difficult
- scalability

Remembering controllers: (Perceive && Memory) $\rightarrow$ State $\rightarrow$ Action

- Game state = \{ perceptions \} + \{ memories \}
- memories = \{ fn(moves) \} // memory changes game state
- probabilistic decisions

Learning controllers:

- machine learning
- reinforcement (habit, strength)
Graphs: FSM, distributed NPC history

FSM represents current state of NPC and possible transitions (productions) to other states.

Graphs can store information about history of NPC activity around node.
Use to decide next behavior.
Remembering

(Perceive & Memory) ➔ State ➔ Action

Game state = \{ perceptions \} + \{ memories \}
memories = \{ fn(moves) \}

Action can change the game state – change values of a remembered attribute.

Memory states (values) valid? How often are values validated.
forgetting – probability of recall decreases w/ time?

Memory state is a strategy (explore, hide), an emotion (rage, fear), ...

Memory state value transitions – FSMs
gameState determines allowable changes of current memory value.

Production rules can change memory values.

Probabilistic "rules" can change memory values.
Learning

Simulated learning – all NPC knowledge is known "in game" but revealed (used) as game time progresses – NPC appears to learn.

Learning NPC decides action based on state, action has a reward and a utility.

Utility = immediate reward (+, 0, -) + long term reward (heuristic)

reward evaluated at time of action
utility of action can be modified by later actions (steps)

S-R learning has no utility – decisions with higher reward are learned.

State whose action has highest reward are chosen next

Learning could:

re-order a Production System
change probability (or recalling) memory states
change costs (utility) of way points in path finding
Blending

Blending – compositing, adds values from two pixels to determine the rendered pixel value.

\[ \text{NewColor} = \text{SrcBlend} \ast \text{SrcColor} + \text{DestBlend} \ast \text{DestColor} \]

Both SrcBlend and DestBlend can be defined.

By default, MonoGames expects the images to contain transparency information for each pixel in the A (Alpha) channel, which contains the transparency information.

Image formats (like PNG) store this information, other (like JPG) don’t. DirectX meshes have material color values.

Modelers can set RGBA properties of materials (colors).

Fog is a built-in blending technique using distance from camera.
Effect – shaders

Effect represents shader effects enables effect query and selection requires an effect.fx file

Add shader file to Contents subdirectory
Load the effect file
Set effect 's parameters
Set effect 's technique

Draw model with effect (just like w/ BasicEffect)

Every vertex that is drawn will pass through the vertex shader.

Every pixel drawn will pass through the pixel shader.

HLSL is a C like language stored in a separate file (*.fx)
uniform float4x4 WorldViewProj : WORLDVIEWPROJECTION;

struct VS_OUTPUT {
    float4 position : SV_POSITION;
    float4 color : COLOR0;
};

VS_OUTPUT VertexShaderFunction( 
    float4 Pos  : SV_POSITION,
    float4 Color : COLOR0 ) {
    VS_OUTPUT Out = (VS_OUTPUT)0;
    Out.position = mul(Pos, WorldViewProj);
    Out.color = Color;
    return Out;
}

float4 PixelShaderFunction( VS_OUTPUT vsout ) : COLOR {
    return vsout.color;
}

technique TransformTechnique {
    pass P0 {
        vertexShader = compile vs_4_0 VertexShaderFunction();
        pixelShader = compile ps_4_0 PixelShaderFunction();
    }
}
Ambient and Diffuse light

Ambient light intensity ($I_a$) at a surface point has been scattered by many reflections and is the source light's ambient color ($L_a$) scaled by surface's ambient (material) reflectivity constant ($K_a$).

$$I_a = L_a * K_a$$

Diffuse light intensity ($I_d$) at a surface point is a function of the source light intensity ($L_d$), the point's normal ($n$) to the surface, and the surface's diffuse reflectivity constant ($K_d$).

$$I_d = L_d * K_d * \text{dot(surfaceNormal, -lightSource)}$$

Ambient and Diffuse lights are not affected by the position of the viewer. Diffuse light is omni-directionally reflected.
Pixel shader – light attenuation

Point (bare bulb) and spot lights (flashlight) intensity decreases with distance of light from surface.

Point light has position but no direction.

Spot lights have position, direction, an inner radius (maximum) and outer radius (falloff) of light attenuation.

Assume vertex shader output includes vertex position and normal.

Pixel shader computes light intensity to add to pixel.

Point light's contribution to pixel intensity

\[ \text{LightFactor} = \text{dot}(\text{normal}, - \text{lightPosition}) \]
Dual Display Stereo Viewing

Scene is rendered from as a view from each eye (often separate buffers). Position, orientation, aspect ratio (X to Y), parallax (inner pupilary distance), convergence, convergence distance (perceived stereo), distance L R eye.

Convergence angle often not implemented...

![Diagram with labels: hither, convergence, parallax]
Head tracking

Head tracking with camera (regular || IR)
tracking targets
face recognition (eyes, shape of head)

Search image (for pixel targets) / frame

3 tracking targets

- calibrate, store size and vector lengths of targets: b'.size, bc'.vec
- volume of target $\Leftarrow$ change in distance
- vectors between targets $\Leftarrow$ change in orientation

$$b\.size > b'\.size \&\& c\.size < c'\.size \&\& bc\.vec < bc'\.vec \&\& a\.size < a'\.size \Rightarrow \text{head has turned (Y – yaw) right}$$

magnitude of the rotation is proportional to changes
need to determine expressions and set weights in calibration

face recognition
- eyes are "targets" (whites w/ pupil pattern) w/in field (head)
What to do next

Examine and practice with class concepts in more depth.

Programming  (see class page “resources” link)
  Animating hierarchical meshes
  Shaders – DirectX effects, alpha blending
  Textures – usage, coordinate transforms, programming
  DirectX 11 – C++, HLSL geometry shader functions
  OpenGL – C++, GLSL, WebGL, OpenGL-ES
  Algorithms – BSP / Quad / Oct trees, path finding,
    Game Intelligence, procedural modeling system,
    L-systems (L Parser, L Studio...)
  Frameworks / Engines – SharpDX, VTK, Unity…
  GPU programming – CUDA, OpenCL

Modeling tools – learn or practice more w/ AC3D, Blender

Visualization – learn more about visual thinking, art

Participation – SIGGRAPH (in LA 7/30 – 8/4), IGDA