CS 480/680: GAME ENGINE PROGRAMMING

GRAPHICS: 2D AND 3D RENDERING

1/24/2013
Santiago Ontaño
santi@cs.drexel.edu
Outline

• Student Presentations
• Primitive Rendering
• Scenes
• Project Discussion
Outline

• Student Presentations
• Primitive Rendering
• Scenes
• Project Discussion
Student Presentations

• Alexander Duff:
  • “Fast Collision Detection for 3D Bones-Based Articulated Characters”

• Colin Mckenna:
  • “A Flexible Simulation Architecture for Massively Multiplayer Games”

• Hans Formon:
  • “Fractal Terrain Generation”
Outline

• Student Presentations
• Primitive Rendering
• Scenes
• Project Discussion
Game Engine Architecture

- Game Specific
- Game Engine Functionalities
- Resource Management
- Utility Layer
- Platform Independence Layer
- SDKs
- OS
- DRIVERS
- HARDWARE

Game Engine Dependencies
Game Engine Architecture

Rendering Engine

Gameplay Foundations (Game Loop)

- Scripting
- Artificial Intelligence
- Online Multiplayer

- Animation Engine
- Physics
- Audio Subsystem
- Profiling & Debugging

- Collisions
Rendering Engine Architecture

- Front End
- Visual Effects
- Scene Rendering
- Rendering Primitives
- Text
Rendering Engine Types

2D Bitmaps graphics:
Most classic games (Mario, etc.)

3D Vector graphics:
Most modern games
## Rendering Engine Types

<table>
<thead>
<tr>
<th></th>
<th>Bitmap (discrete)</th>
<th>Vector (continuous)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>2D</strong></td>
<td>Most classic games (Mario, etc.)</td>
<td>Asteroids, Thrust, etc.</td>
</tr>
<tr>
<td><strong>3D</strong></td>
<td>Voxel rendering (comanche, Voxatron, etc.)</td>
<td>Most modern games</td>
</tr>
</tbody>
</table>
Rendering Engine Types

2D Bitmap graphics
Rendering Engine Types

2D Vector graphics
Rendering Engine Types

3D Bitmap graphics
Rendering Engine Types

3D Vector graphics
Basics of Rendering

• Whatever your internal representation of the graphics of your game (2D, 3D, bitmaps, vectors), rendering means “projecting those graphics to the frame buffer”
• Frame buffer is the portion of memory that stores the data that will be display by your computer screen (it’s basically an array)
Basics of Rendering

- Whatever your internal representation of the graphics of your game (2D, 3D, bitmaps, vectors), rendering means "projecting those graphics to the frame buffer."

- Frame buffer is the portion of memory that stores the data that will be displayed on your computer screen.

- The frame buffer is basically an array of bytes, where (typically) each 3 bytes is a pixel (R, G, B).
  - You can write to it directly as if it were any other array (old school).
  - You can use the hardware on your graphics card to render to it fast.

Rendering Engine

Frame buffer
Basics of Rendering

- Your graphics card will send the content of the frame buffer **automatically** 60 times per second (depending on the refresh rate you set for your monitor).
- You need to render each frame of the game in sync with this (this is called “Vertical Synchronization”, or “V-sync”).
- Typical solution: double buffering
Basics of Rendering

- Your graphics card will send the content of the frame buffer automatically 60 times per second (depending on the refresh rate you set for your monitor).
- You need to render each frame of the game in sync with this (this is called "Vertical Synchronization", or "V-sync").
- Typical solution: double buffering.
Basics of Rendering: Double Buffering

- Have 2 (or more!) screen buffers. While the monitor is displaying one, the game engine is rendering in the other.
- Once you are done, swap buffers!
For Your Project

• Manually handling double buffering is very old school, modern graphic libraries do it for you.

• For example, in Java:
  • `this.createBufferStrategy(2);` // call this from your JFrame class
  • Java will flip the buffers each time the paint method gets called

• In C++ using SDL:
  • Initialize SDL with double buffer:
    • `SDL_SetVideoMode( … , SDL_DOUBLEBUF);`
  • After rendering, call:
    • `SDL_Flip(screen_surface);`  // this will trigger the flip

• In C++ using SDL + OpenGL:
  • After rendering, call:
    • `SDL_GL_SwapBuffers();`
For Your Project

Game loop:

While(!quit) {
    listen events from the OS
    speed control
    game cycle
    Render

    swap_buffers()

    yield some CPU to the OS (e.g. “SDL_Delay(1)” in C++, or “Thread.sleep(1)” in Java)

}
Rendering Engine Architecture

Rendering Engine

Front End

Visual Effects

Scene Rendering

Rendering Primitives

Text
Rendering 2D Bitmaps

• In classic 2D games, all images are “sprites” (i.e. 2D bitmaps)

• A Sprite may have:
  • A **bitmap**: a rectangular matrix that stores the color and transparency of each pixel (typically just an array of bytes, where each 4 bytes is a pixel: RGBA).
  • **“hot spot”**: the x, y coordinates that are considered the center of this sprite
Rendering 2D Bitmaps

- The hot-spot has many uses. For example, making sure different sprites in an animation flow correctly.
Rendering 2D Bitmaps

- The hot-spot has many uses. For example making sure different sprites in an animation flow correctly.

When playing an animation consisting of a series of sprites, they are drawn making sure the hot-spot remains in the same pint in the screen. In this way, graphic artists can control any necessary offset that is required for the animation.
In Your Project

- The primitive rendering module for a 2D bitmap game only needs to support sprite drawing (unless you want to add some fancy effects to your engine, more on this later)
- Maybe you also want to add “pixel drawing” as one of your primitives for effects.
- A typical Sprite class:

```java
Class Sprite {
    Bitmap bm;
    int hot_x;
    int hot_y;

    void draw(int x, int y) {
        PrimitiveRenderer.renderBitmap(bm, x - hot_x, y - hot_y);
    }
}
```

I’m assuming you know how to render bitmaps in the language of choice for your projects. SDL, Java2D, Javascript, etc. give you this. If you use OpenGL, you need to transform the bitmap to a texture, and then draw a square with that texture.
In Your Project

• The primitive rendering module for a 2D bitmap game only needs to support sprite drawing (unless you want to add some fancy effects to your engine, more on this later)
• Maybe you also want to add “pixel drawing” as one of your primitives for effects.
• A typical Sprite class:

```plaintext
Class Sprite {
    Bitmap bm;
    int hot_x;
    int hot_y;

    Void draw(int x, int y) {
        PrimitiveRenderer.renderBitmap(bm, x - hot_x, y - hot_y);
    }
}
```

Additional features like rotation or scaling could be added.
Rendering 2D Vector Graphics

• Primitives:
  • **Pixels**: most graphic libraries support this. If you want to go old school, you can manually access the frame buffer for this.
  
  • **Lines**: all graphic libraries support this. Otherwise, if you want to go old school, “Bresenham Algorithm” is what you need to draw lines.
  
  • **Ovals**: same here (and there is also a “Bresenham Oval Algorithm”)
  
  • **Boxes**: boxes are trivially drawn using a pair of nested for loops if your graphics library does not support it.
  
  • **Polygons**: these are more complex to draw (specially if you want to allow filling of concave polygons). But there are algorithms: “scan-line polygon filling algorithm”.
Rendering 3D Vector Graphics

- There are many 3D primitives you could support (lines, points, triangles, boxes, spheres, cylinders, etc.). However, for games, you typically just need two:
  - **Triangle Meshes**: to render any 3D object specified as a collection of triangles (e.g. edited with Maya)
  - **Flat 2-dimensional boxes**: (to be used to render any 2D graphic we might want to overlay, such as text)
Rendering 3D Vector Graphics

• Data structure for the mesh primitive:
  • A list of triangles
  • Each triangle should store:
    • Normal (can be computed from the triangle, but it’s good to cache it)
    • Texture or color (plus texture mapping coordinates)

• The functionality you should support is:
  • Draw the primitive
  • Draw it given an offset (x,y,z)
  • Draw it given a rotation (quaternion): more on this later!
  • Draw it given an offset and a rotation
Rendering 3D Vector Graphics

• Data structure for the mesh primitive:
  • A list of triangles
  • Each triangle should store:
    • Normal (can be computed from the triangle, but it’s good to cache it)
    • Texture or color (plus texture mapping coordinates)

• The functionality you should support:
  • Draw the primitive
  • Draw it given an offset (x,y,z)
  • Draw it given a rotation (quaternion)
  • Draw it given an offset and a rotation

I’m assuming you know how to render triangles and textures in your language of choice. If you don’t (and no one in your project focuses on graphics), then use an off-the-shelf library for rendering (ask me if you need help with this!).
Rendering 3D Vector Graphics

- Other courses at Drexel cover this in detail (depth buffer, etc.)
- Not going to spend much time on it:
  - OpenGL allows you to draw colored or textured triangles
  - Should be easy to write the code to draw meshes using those functions.
- The most complex part is to obtain the models: code to load models from disk into your data structures.
  - I recommend the library “Open Asset Importer” if you use C++, it can import many 3D formats, for which you can find available models online.
  - In the examples of OAI, you will also find code to draw the meshes in OpenGL.
Rendering 3D Bitmap Graphics (voxels)

- Two basic rendering techniques:
  - Use a 3D vector graphics backend (like Voxatron): [http://www.youtube.com/watch?v=EKdRri5jSMs](http://www.youtube.com/watch?v=EKdRri5jSMs)
    - Just consider each voxel as a 3D cube
    - This is good for low-resolution voxels, it produces a nice retro-effect.
  - Ray-tracing (this can be used for 3D Vector graphics too, but, as of today, it’s too slow to run on real time, except on specialized hardware):
    - Better for high-resolution voxel models

- Primitive Rendering module needs same functionality as 2D bitmap one: drawing sprites (this time, they are 3D sprites with voxels, instead of pixels).
Voxel Model

• Each Voxel can be rendered as a cube, or as any other primitive (a sphere, rounded-edge cube, etc.)

• Voxel graphics:
  • Stored in 3D matrices: each position containing R,G,B,A of each voxel

• There are available editors, e.g.: Sprooxel (free)
Rendering 3D Bitmap Graphics (voxels)

- Raytracing:
Rendering 3D Bitmap Graphics (voxels)

- Raytracing:

The area seen in the screen can be considered as a plane that sits in between the camera, and the scene to be rendered.
Raytracing:

For each pixel in the screen, we can define a “ray” as:
- The line that intersects both the camera and the center of the pixel
Rendering 3D Bitmap Graphics (voxels)

- **Raytracing:**

  We can simulate the path light would take to reach the camera if coming from the direction of this “ray”:
  For example, computing the closest intersecting object in the scene, we know what is the object we will see in that pixel.

  Once we know the point in the object that we need to render, we can compute which light sources affect this point, in order to calculate the final R,G,B value for that pixel.

  *(more on lighting later)*
Rendering 3D Bitmap Graphics (voxels)

• Raytracing:

For Voxels, this might be enough (we compute the closest voxel). But we can do more than this: if the surface of the object can reflect light (e.g. a mirror), we can further follow the path of light to see what would be seen in the mirror in that location.

Raytracing is computationally expensive, but can produce very realistic results.
Rendering Engine Architecture

Rendering Engine

- Front End
- Visual Effects
- Scene Rendering
- Rendering Primitives
- Text
Rendering Fonts

- There are two typical approaches to render text in 2D games:
  - TTF fonts:
    - Defined by their geometry (just their shape).
    - Can be rendered in any color, and size
  - Bitmap fonts:
    - Defined similarly to any other sprite in a 2D game
    - Each letter is a bitmap
    - Cannot be scaled as nicely as TTF fonts
    - Can be used to define textured, or pixelated fonts
Rendering Fonts: TTF Fonts

Example libraries:
- Java / Javascript: this feature is built-in
- C++: SDL_ttf

Sample output:
Rendering Fonts: Bitmap Fonts

- You will handle them as standard sprites. Create a large bitmap like this:
- Each letter should be in the position corresponding to its ASCII code.
- To render a line of text, you just look for the image corresponding to the ASCII code of each letter.
Rendering Fonts: Bitmap Fonts

• You will handle them as standard sprites.
• Create a large bitmap like this:
• Each letter should be in the position corresponding to its ASCII code.
• To render a line of text, you just look for the image corresponding to each letter's ASCII code.
Rendering 3D Fonts (3 choices)

• Textured Bitmap/TTF fonts:
  • Render a 2D font as a texture, and apply it to a 3D shape

• Extrude a 2D TTF font:
  • You can do it by hand from a TTF
  • Library: FTGL

• Model a 3D font (I’ve never seen it in a game):
  • Vector: author meshes (e.g. with Maya) for each font
  • Voxel: define 3D bitmaps for each character
Outline

- Student Presentations
- Primitive Rendering
- Scenes
- Project Discussion
Rendering Engine Architecture: 2D
Scene Rendering

- Requires:
  - Transformations (rotations, translations)
  - Composition of primitives (to create complex game objects, or maps)

- They are the same for bitmap/vector graphics
Transformations

- The basic transformations are:
  - Translation
  - Rotation
  - Scaling

- They can all be represented as transformation matrices
  - To combine more than one transformation, you just multiply matrices. So, you can have a complex sequence of transformations all collapsed into a single matrix
2D Transformations

- Homogeneous coordinates: \((x,y,1)\)

- Translation:

\[
\begin{bmatrix}
1 & 0 & dx \\
0 & 1 & dy \\
0 & 0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
1 \\
\end{bmatrix} =
\begin{bmatrix}
x + dx \\
y + dy \\
1 \\
\end{bmatrix}
\]

- Rotation:

\[
\begin{bmatrix}
\cos(a) & \sin(a) & 0 \\
-sin(a) & \cos(a) & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\]

- Scaling:

\[
\begin{bmatrix}
sx & 0 & 0 \\
0 & sy & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\]
3D Transformations

- Homogeneous coordinates: \((x, y, z, 1)\)

- Translation:

\[
\begin{bmatrix}
1 & 0 & 0 & dx \\
0 & 1 & 0 & dy \\
0 & 0 & 1 & dz \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

- Rotation:

\[
\begin{bmatrix}
cos(a) & -sin(a) & 0 & 0 \\
sin(a) & cos(a) & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

This rotates around the z axis, similar matrices exist for X or Y axis.

- Scaling:

\[
\begin{bmatrix}
sx & 0 & 0 & 0 \\
0 & sy & 0 & 0 \\
0 & 0 & sz & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]
Representing Game Objects

- Game objects (characters, maps, levels, etc.) are constructed by combining "primitives" and "transformations"
- For example:
  - Imagine you have a collection of sprites for the body parts of a character
  - You can combine them together with rotations and translations
Representing Game Objects
Representing Game Objects

Class GameObject {
    Primitive p;
    int nsubParts;
    Transformation []part_transformations;
    GameObject []part;

    void draw(Transformation t) {
        ...
    }
}
Representing Game Objects

• This only accounts for rendering
• The GameObject data type can be extended with:
  • Collision data
  • AI data (behavior)
  • Animation data

• An alternative (if you want to keep the different modules as independent as possible):
  • Each module keeps their own GameObject data type (RenderingGameObject, AnimationGameObject, CollisionGameObject, etc.)
  • Higher level GameObject simply contains one of each.
Representing Maps

- 2D bitmap games:
  - Most typical are tile-maps:
    - A tile-map is an integer 2D array
    - Each integer represents a “tile”
    - Each “tile” is a Sprite or GameObject
    - Each one has properties:
      - Walkable/not walkable
      - Etc.
  - Maps can have multiple “layers”
    - Each layer is a tile-map
  - For example:
    - Layer 0 is the background
    - Layer 1 are the items, and characters
    - Layer 2 are overlays (clouds, etc.)
  - For rendering: first render layer 0, then layer 1 on top, then layer 2.
Representing Maps

- **3D Maps:**
  - **Height map:**
    - 2D array where each position represents a height of the terrain
    - Might be paired with a “terrain-type” array for terrain type (water, grass, etc.)
  - **Height mesh:**
    - 2D polygonal partition of the map, representing the height of each point (this is what was used in the original DOOM)
Representing Maps

• 3D Maps:
  • Full meshes:
    • Most versatile
    • Problem: collision detection-intensive (can be accelerated by preprocessing, some student presentations about that)

• Voxel maps:
  • Same as 2D bitmap maps, but with 3 dimensional arrays, instead of 2 (and no need for layers).
A Complete Scene Renderer

Front End

Visual Effects

Scene Rendering

Rendering Primitives

Text

Game State

Map

Camera

Game Object

Game Object
Camera is basically a transform that is applied by scene rendering so all objects appear as if seen from the point of view of the camera.

It can be represented as:
2D: map, center(x,y), angle, scale
2D: map, origin, direction, orientation
A Complete Scene Renderer

Basic method:

renderScene(Camera c, GameState gs)

Figures out the necessary transforms given the camera, and calls the primitive rendering to draw the map and all the game objects.
Rendering Engine Architecture: 2D
Lights

- Your scene might contain lights
- Lights specified in the map

Lights might imply:
- Color change (colors change depending on light):
  - This is automatically given to you by libraries like OpenGL (but using a very basic lighting model).
  - You can improve on top of it (check “Phong” method).
- Shadows (this is more complex, student presentation about this).

Either in 2D or in 3D, lights require:
- Computing the angle and distance between the light source and the surface being illuminated
- Computing the areas/volumes not blocked from light (for shadows)
Lights
Other Effects:

• Motion blur: achieved by internally storing the position of all the objects in previous frames, and rendering them with some transparency, before rendering the new scene.

• Glow: typically for vector graphics
  • Can be done by hand
  • Easier with a shader
  • I’m not explaining shaders! 😊
Rendering Engine Architecture: 2D
Outline

- Student Presentations
- Primitive Rendering
- Scenes
- Project Discussion
Links to Game Videos

• Two interesting games:
  • Ace of Spades:  
    http://www.youtube.com/watch?v=RK5utVkjOi4&feature=youtu.be  
  • Mari0:  
    http://www.youtube.com/watch?v=SaoHMjS04vU
  • Not Tetris 2:  
    http://www.youtube.com/watch?v=8edwWVSHsrY
Potential Project Topics on Rendering

- Voxel engine
- Shadow casting
- 2D visual effects:
  - Motion blur / glowing effects / particle systems / lighting
- Full-fledged 3D rendering engine (not using any library)
- Many more, be original!

- If you go for a “graphics topic” on 2D bitmap engine:
  - 2D bitmap rendering is very simple (all graphics libraries support this), so you’ll need to do something interesting on top (lights, effects, or code the rotations/scaling yourself, without the use of the library, etc.)
What if no one in your group does graphics?

- Options:
  - 2D bitmap engine (a basic one can be coded very easily)
  - Basic 3D rendering engine without shadows, textures, etc. just use a library to load models, and render them using OpenGL
Separation between Game and Game Engine

• Example:
  • Project 2D RTS game in Java
  • Student 1 focus: AI
  • Student 2 focus: networking
  • Demo: simple RTS game with 4 unit types (bases, workers, melee units, ranged units)

• Where does the game engine end and the game starts?
Separation between Game and Game Engine

- Rendering Engine
- Scripting
- Artificial Intelligence
- Online Multiplayer
- Gameplay Foundations (Game Loop)
- Animation Engine
- Physics
- Audio Subsystem
- Collisions
- Profiling & Debugging

Separation between Game and Game Engine
Separation between Game and Game Engine

- Demo Front-end
- Demo Scripts
- Demo Game Flow

- Scripting
- Artificial Intelligence
- Online Multiplayer

- Game Loop
- Physics
- Audio Subsystem

- Front-end
- Scene Rendering
- Primitive Rendering

- Animation Engine
- Collisions

- Game State

- Resource Management
Separation between Game and Game Engine

- Front-end
- Demo Front-end
- Demo Scripts
- Demo Game Flow
- Online Multiplayer
- Game Loop
- Physics
- Collisions
- Audio
- Scene Rendering
- Primitive Rendering
- Resource Management
- Sprite class on top of Java 2D rendering capabilities (text and bitmaps)
- Game State
- Sprites
- Unit definitions
- Maps

- Extremely simple physics and collision (basically just collision between boxes)

3 resource managers:
- Sprites
- Unit definitions
- Maps
Separation between Game and Game Engine

Demo Front-end
Demo Scripts
Demo Game Flow

Scripting
Artificial Intelligence
Online Multiplayer

Game Loop

Scene Rendering
Primitive Rendering

Traversing game state and render map and units

Map and GameObject classes, player info

3 resource managers:
- Sprites
- Unit definitions
- Maps

Resource Management
Very simple support for scripting: maybe just add an empty class “Script”, with an abstract method “execute”, so they can be defined in the demo.
Separation between Game and Game Engine

Focus of student 1: networking to allow n players

Focus of student 2: Advanced path-finding techniques (e.g. k-NN-LRTA*)

Demo Front-end
Demo Scripts
Demo Game Flow

Scene Rendering
Primitive Rendering

Resource Management

Game State
Game Loop
Physics
Collisions
Artificial Intelligence
Online Multiplayer

Scripting
Separation between Game and Game Engine

- **Demo Front-end**
- **Demo Scripts**
- **Demo Game Flow**
- **Scripting**
- **Artificial Intelligence**
- **Online Multiplayer**
- **Audio Subsystem**
- **Game State**
- **Collisions**
- **Resource Management**

Your game loop, should provide a “hook” (e.g. a call-back, or a simple method you can redefine) to allow for your personalized front-end rendering. Which is specific of the demo.

Necessary scripts for the 4 units in the demo (the game flow code will assign them adequately to their respective units on startup).

FSM (as seen last week) for the game flow
Separation between Game and Game Engine

• Summary:
  • Game engine provides general functionalities that could be used to define any RTS game (with some limits, of course):
    • General unit definition (through some configuration file, you assign movement speed, resources needed, graphics, etc.)
    • General asset loading (graphics, maps, etc.) in a general file format
    • General rendering
    • General pathfinding
    • General networking capabilities (independent of the game you create)

  • Then, for the demo:
    • You create the configuration files (types of units, etc.)
    • Get the assets (graphics, maps)
    • Write the game-flow and any other game-specific code you need (e.g. front-end)

  • Advise:
    • Separate the code: for example, have 2 projects, one for game engine, one for game. Game engine should be able to compile on its own.
Remember that next week:

• First Project Deliverable:
  • Groups
  • Topics
  • Specific algorithms
  • Specific libraries you will use
  • Specific structure of your game engine (not necessarily a class diagram, see the diagrams shown in class)
  • Demo you will create

• Submission procedure:
  • Email to (copy both):
    • Santiago Ontañón santi@cs.drexel.edu
    • Stephen Lombardi sal64@drexel.edu
  • Subject: CS480-680 Project Deliverable 1 Group #