Arcade Games:
2D Bit-mapped Graphics

John Laird and Sugih Jamin

September 11, 2006
2D Graphics

• Most 2D games use sprite instead, but basic primitives applicable in 3D graphics
• Points
  • x, y
• Lines
  • Two points
  • Draw by drawing all points in between
  • Low-level support for this in hardware or software
Coordinate System

(0,0)  +x

(120,120)

+y
Polygons

• Defined by vertices
• Closed: all lines connected
• Draw one line at a time
• Can be concave or convex
• Basis for many games
• Basis for 3D graphics (triangle)

• Required data:
  • Position: x, y
  • Number of vertices
  • List of vertices
  • Color (shading)
Operations on Polygon

- Translation: moving
- Scaling: changing size
- Rotation: turning
- Clipping and scrolling
Translation: Moving an Object

• To move an object, just add in changes to position:
  • \( x = x_0 + dx \)
  • \( y = y_0 + dy \)

• If have motion, the \( dx \) and \( dy \) are the x and y components of the velocity vector.

\[ \begin{align*}
  dx &= \cos v \\
  dy &= \sin v 
\end{align*} \]
Positioning an object

• Problem: If we move an object, do we need to change the values of every vertex?

• Solution: change frame of reference
  • *World* coordinate system for object positions
    • coordinates relative to screen
  • *Local* coordinate system for points in object
    • coordinates relative to the position of the object (frame of reference)

Triangle location: 4,0

P1: 0, 1
P2: -1, -1
P3: 1, -1
Scaling: Changing Size

- Multiply the coordinates of each vertex by the scaling factor.
- Everything just expands from the center.
  - `object[v1].x *= scale`
  - `object[v1].y *= scale`
Rotation: Turning an Object

• Spin object around its center in the z-axis.
• Rotate each point the same angle
  • Positive angles are clockwise
  • Negative angles are counterclockwise
• \( x = x_0 \times \cos(\text{angle}) - y_0 \times \sin(\text{angle}) \)
• \( y = y_0 \times \cos(\text{angle}) + x_0 \times \sin(\text{angle}) \)
• Note: angle measured in radians not degrees!
Matrix Operations

- Translation, rotation, scaling can all be collapsed into matrix operations:

- Translation:

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

- Scaling:

\[
\begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}
\]

- Rotation:

\[
\begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} \cos & -\sin & 0 \\ \sin & \cos & 0 \\ 0 & 0 & 1 \end{bmatrix}
\]

\(sx, sy = \) scaling values
Putting it all together

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>1</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>sx*\cos</td>
<td>-sx*\sin</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>sy*\sin</td>
<td>sy*\cos</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dx</td>
<td>dy</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Common Problems: Flicker and Tearing

- Video update slower than display
- Change video buffer during updating

Solution:
- Double buffering -- write to a “virtual screen” that isn’t being displayed.
- Either BLT buffer all at once, or switch pointer.
Clipping

• Display the parts of the objects on the screen
  • Can get array out of bound errors if not careful
  • Easy for sprites – done in DirectX

• Approaches:
  • Border vs. image space or object space
Image Space vs. Object Space

• **Image space:**
  • What is going to be displayed
  • Primitives are pixels
  • Operations related to number of pixels
    • Bad when must do in software
    • Good if can do in parallel in hardware – have one “processor”/pixel

• **Object space:**
  • Objects being simulated in games
  • Primitives are objects or polygons
  • Operations related to number of objects
Border Clipping

- Create a border that is as wide as widest object
  - Only render image, not border
  - Restricted to screen/rectangle clipping
  - Still have to detect when object is outside border
  - Requires significantly more memory
Image Space Clipping

- **Image Space**:
  - The pixel-level representation of the complete image.

- **Clipping**
  - For each pixel, test if it is inside the visible region
  - If buffer is 320x200, test 0-319 in x, 0-199 in y.

- **Evaluation**
  - Easy to implement
  - Works for all objects: lines, pixels, squares, bit maps
  - Works for subregions
  - Expensive! Requires overhead for every point rendered if done in software
  - Cheap if done in hardware (well the hardware cost something)
Object Space Clipping

• Object space:
  • Analytical representation of lines, polygons, etc.

• Clipping
  • Change object to one that doesn’t need to be clipped (e.g., shorten the line)
  • New object is passed to render engine without any testing for clipping

• Evaluation
  • Usually more efficient than image space software
    • But hardware support of image space is fast
  • Need different algorithm for different types of objects
    • Lines are easy. Concave objects are problematic
    • Usually just worry about bitmaps
Line Clipping Cases
Arcade Games

• Examples
  • Missile Command, Space Invaders, Breakout, Centipede, Pac-Man, Frogger, Tempest, Joust,

• Important Traits:
  • Easy-to-learn – simple controls
  • Move objects around the screen
  • Single-screen – or simple scrolling
  • Infinite Play
  • Multiple Lives
  • Scoring – highest score
  • Little to no story
Game Loop

Initialization → Overall Game Control → Game Session Control → Player Input → Main Logic
- Physics
- Game AI
- Collision Processing

Render scene to buffer → Copy buffer to display → Time sync → 1 cycle/frame → wait → Exit

Overall Game Control
- Game Session Control
- Player Input
- Main Logic
- Render scene to buffer
- Copy buffer to display
- Time sync
- wait
Static Objects

- Background, frame, fixed building, maze structure, …
- Draw only once
- Can be very complex
Dynamic Objects: Sprites

Usually small number of pixels
Most be draw on screen 30 times/second
• Save background that sprite covers
• Player’s Sprite
  • Paddle, gun, tank, …
  • User can move it, turn, shoot, …
• Game Sprites
  • All of the other objects in the game that move
  • Bullets/missiles shot by player
• Most common interaction is collision
  • Fast collision detection is important
Sprites:

• Object that moves around, displayed as a bit map
• \( N \times M \) pixels: \( 12 \times 12 = 144 \). \( 100 \times 100 = 10,000 \).
• Displayed on a background
Sprite Data

• Static
  • Size
  • Image sets
  • Weapons, shields, worth, ...

• Dynamic
  • Position
  • Velocity
  • Pose
  • Current image
  • Strength, health, ...
  • Saved background
Creating Sprites

• Create Sprite in 2D or 3D drawing package
  • 2D
    • Gimp
    • Photoshop
    • Corel’s Paint Shop Pro (was JASC) or Painter (was Fractal Design)
  • 3D
    • Blender 3D
    • Milkshape 3D
    • 3D Studio Max
    • Maya

• Save as file
Drawing the Sprite

• Some parts of the sprite are transparent
  • Use a special code (255) to be transparent
  • When drawing the pixels, don’t copy that code
  • Is expensive because done for every pixel

• Some sprites have no transparencies
  • Can have separate draw function
  • Avoid test for transparency
Sprite Movement and Display

• Compute new position of Sprite
• If Sprite moved, erase Sprite by restoring saved background
• Save background where Sprite will go
• Draw Sprite
Run-Length Encoding

• Compress Sprites in files using “run-length encoding” (RLE).
  • Instead of representing every pixel, encode number of consecutive pixels of same kind in a row
  • Big win if lots of same color in a row (transparent)
  • Doesn’t capture vertical or 2D structure well.

• Not so good:

• Much better:
Sprite Scaling

• Used to show change in depth (distance)

• Options:
  • Dynamic computation
    • Can lead to very blocky pictures when they get big
  • Pre-store different sizes
    • Hard to get large numbers of intermediate sizes
  • Pre-store different sizes for major size changes: x2
    • Dynamically compute intermediate sizes

• Supported in Direct-X (in hardware and software)
Depth

- Can fake depth by scaling but what if overlap?
  - Want closer objects to cover distant objects
  - Associate depth with each Sprite - usually small number

- Image space solution
  - Maintain shallowest depth rendered
  - Add pixel if closer than previous
  - Lots of work at each pixel if in software
  - Hardware Z-buffer to rescue - standard for game machines

- Object space solution
  - Sort objects by depth
    - $O(#\_of\_objects \times \log(#\_of\_objects))$
  - Draw back to front
Sprite Rotation

• Store each orientation as a separate bit map
  • 16 different pictures is reasonable start

• Pick the closest one to the current orientation

• Calculating from scratch usually too slow
• Sometimes supported by hardware
Sprite Animation

• Changes in the display as state of object changes
  • Example: standing, sitting, jumping, singing, shooting

• Choose the current bit-map based on object state
  • Might require separate timer for animation changes

• Storage if including rotation
  • \(_{\text{num}}_{\text{bitmaps}} = \_{\text{num}}_{\text{angles}} \times \_{\text{num}}_{\text{states}}\)
Semi-static Objects

• Rarely changes, doesn’t move
• Examples: Walls that can be damaged
• Change drawing on screen or buffer
• Not worth redrawing every cycle
• Do not have to save background
Dynamic Background

• If the background is scrolling or changing a lot
  • Redraw complete buffer from scratch
  • Avoid saving background for sprites
  • More drawing

• Either
  • Draw from back to front
  • Draw using z-buffer or z-list
Scrolling - simple

Horizontal scrolling: usually side view of world
Scrolling - simple

Vertical scrolling: usually top view of world
Scrolling – Tile Based

Tile map

screen
Scrolling – Sparse

• Object-based
  • Keep list of objects with their positions
  • Each time render those objects in current view
  • Go through list of object – linear in # of objects

• Grid-based
  • Overlay grid with each cell having a list of objects
  • Only consider objects in cells that are in view
Collision Detection

• Image Space:
  • Pixel by pixel basis. Expensive.

• Object Space:
  • Hard for complex and concave spaces:

• Standard Approach:
  • Cheat!
  • Create a bounding box or circle
    • test each vertex to see in another object
  • Hide this by making your objects boxy
  • Don’t have objects like:
    • Can use multiple bounding shapes and bounding areas
Sprite Collisions

• Easiest:
  • Use the bounding box that includes all the pixels
  Test if vertex of one in bounding box of other

• Tricky:
  • Use something a little smaller to avoid some fake collisions
  • If things happen fast enough, people can’t tell

• Almost right but expensive:
  • Test if non-transparent pixels overlap
  • Can still miss some cases...
Collision?

Be extra careful if variable time step is used in game loop