2D Graphics
John E. Laird
Based on “Tricks of the Game-Programming Gurus” pp.72-109
Lots of obvious observations that make drawing easy

History of 2D graphics: Vector
- Example: Asteroids, Battlezone
- Advantages:
  - Control the electronic gun directly
  - Only draw what is on the screen
  - No jagged lines (aliasing).
  - Only store endpoints of lines
- Problems:
  - Best for wireframes.
  - Must draw everything as lines: text, circles,
  - $SS'$s: Can’t use commercial TV technology
- Example Displays:
  - Textronics, GDP

Raster Graphics
- Advantages:
  - Cheaper
  - Can easily draw solid surfaces
  - Maps screen onto 2D memory
  - Can move blocks of image around, control individual pixels
- Problems:
  - Memory intensive
  - Aliasing problems
- Example:
  - VGA
    - $640 \times 350$ with $16$ colors
    - $320 \times 200$ with $256$ colors

Current Approach
- Use Raster Graphics as underlying technology
  - Memory is cheap
  - Get access is every point on the screen
- Create drawing primitives similar to those in vector graphics
  - Drawing lines
- Support surfaces, textures, sprites, fonts, etc. directly
- Sprites vs. Graphics??

Vector Graphics
- Drawings defined as lines
- Lines stored as endpoints
- Look like wireframes
- No curved lines
- Limited variation in color or intensity.

Raster Graphics
- Screen is made up of “picture elements” = pixels.
- Color defined by mixture of 3-guns: Red, Green, Blue

2D Graphics
- Screen is made up of “picture elements” = pixels.
- Color defined by mixture of 3-guns: Red, Green, Blue
**Translation: Moving an Object**

To move an object, just add in changes to position:
- \( x_{o} = x_{o} + dx \)
- \( y_{o} = y_{o} + dy \)

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

\[
\begin{align*}
\text{Velocity Vector: } V
\end{align*}
\]

\[
\begin{align*}
dx = \cos \theta \\
dy = \sin \theta
\end{align*}
\]

**Scaling: Changing Size**

- Multiply the coordinates of each vertex by the scaling factor.
- Everything just expands from the center.
  - \( \text{object}[v1].x = \text{object}[v1].x \times \text{scale} \)
  - \( \text{object}[v1].y = \text{object}[v1].y \times \text{scale} \)

**Polygons**

- Defined by vertices
- Closed: all lines connected
- Draw one line at a time
- Can be concave or convex
- Basis for many games

- Required data:
  - Number of vertices
  - Color
  - Position: \( x, y \)
  - List of vertices
    - Might be array with reasonable max

**Positioning an object**

- Problem: If we move an object, do we need to change the values of every vertex?
- Solution:
  - \( \text{World} \) coordinate system for objects
    - coordinates relative to screen
  - \( \text{Local} \) coordinate system for points in object
    - coordinates relative to the position of the object

**Coordinate System**

- \( (0,0) \) at origin
- \( +x \) and \( +y \) axes
- \( (120,120) \) point

**2D Graphics**

- Points
  - \( x, y \)
- Lines
  - Two points
  - Draw by drawing all points in between
  - Low-level support for this in hardware or software
**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
- $\text{new}_x = x \cdot \cos(\text{angle}) - y \cdot \sin(\text{angle})$
- $\text{new}_y = y \cdot \cos(\text{angle}) + x \cdot \sin(\text{angle})$
- Remember, C++ uses radians not degrees!

**Matrix Operations**

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

$$
\begin{bmatrix}
  1 & 0 & 0 \\
  0 & 1 & 0 \\
  dx & dy & 1
\end{bmatrix}
\begin{bmatrix}
  x \\
  y \\
  1
\end{bmatrix}
$$

$$
\begin{bmatrix}
  \text{sx} & 0 & 0 \\
  0 & \text{sy} & 0 \\
  0 & 0 & 1
\end{bmatrix}
$$

$$
\begin{bmatrix}
  \cos & -\sin & 0 \\
  \sin & \cos & 0 \\
  0 & 0 & 1
\end{bmatrix}
$$

**Putting it all together**

| $\text{sx}\cdot\cos$ | $-\text{sx}\cdot\sin$ | 0 |
| $\text{sy}\cdot\sin$ | $\text{sy}\cdot\cos$ | 0 |
| $dx$ | $dy$ | 1 |

**Filling in a Triangle: Rasterization**

- Many systems draw 3D objects as collections of triangles -- not arbitrary polygons
- If we can fill a triangle, that is pretty good.
- General idea: draw horizontal lines to fill it in.
- All done in hardware
- Example:

**Fill: Step 1**

- Split triangle into two parts:
  - One flat top
  - One flat bottom
- Gives single slope changes in x as we move vertically

**Step 2 & 3**

- For triangle A, pre-calculate $dx1/y$ and $dx2/y$ based on slopes of edges.
  - Initial $x1$ and $x2$ to x value of vertex.
- Start at top and loop through until $y = 0$
  - Drawing lines from $x1$ to $x2$.
  - Decrement $y$ each time.
  - Subtract $dx1/y$ from $x1$.
  - Add $dx2/y$ to $x2$.
- Inverse for triangle B
Common Problems: Flicker

- Too slow updating
- 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.

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

Speed Issues (Gone)

- Using regular drawing routines
  - Original Microsoft graphics library (GDI) was quite slow
  - Not a problem now – DirectX is ok

- Using Floating Point
  - Floating point used to be much slower than integer
  - Not a problem with Pentium architecture

- Using Standard Trig functions
  - Current machines are fast enough
  - If you start having performance problems, pre-compute and store all rotations you are going to need

Clipping

- Display the parts of the objects on the screen.
  - Can get array errors, etc. if not careful.
  - Easy for sprites – done in DirectX

- Approaches:
  - Border vs. image space or object space

Image Space Clipping

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

- Clipping
  - For each point, test if it is in the region that can be drawn before trying to draw it
  - 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).

Border Clipping

- Create a border that is as wide as widest object
- Only render image
- Restricted to screen/rectangle clipping
- Still have to detect when object is all gone
- Requires significantly more memory
Object Space Clipping
- Object space:
  - Representation of lines, polygons, etc.
- Clipping
  - Change object to one that doesn’t need to be clipped
  - 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

Clipping Filled Triangles
- Do this in image space during rasterization
- Add tests so throw out whole lines – easier because they are all horizontal

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:

Scrolling - simple

Scrolling – Tile Based
- Tile map
### 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