Lecture 14

2D Sprite Graphics
Graphics Lectures

- Drawing Images
  - SpriteBatch interface
  - Coordinates and Transforms

- Drawing Perspective
  - Camera
  - Projections

- Drawing Primitives
  - Color and Textures
  - Polygons

Initial notes:
- bare minimum to draw graphics
- side-scroller vs. top down
- necessary for lighting & shadows
Take Away for Today

- **Coordinate Spaces** and drawing
  - What is screen space? Object space?
  - How do we use the two to draw objects?
  - Do we need any other spaces as well?

- **Drawing Transforms**
  - What is a drawing transform?
  - Describe the classic types of transforms.
  - List how to use transforms in a game.
The SpriteBatch Interface

• In this class we restrict you to 2D graphics
  • 3D graphics are much more complicated
  • Covered in much more detail in other classes
    • Art 1701: Artist tools for 3D Models
    • CS 4620: Programming with 3D models

• In XNA, use the interface SpriteBatch
  • Sprite: Pre-rendered 2D (or even 3D) image
  • All you do is composite the sprites together
Drawing in 2 Dimensions

• Use **coordinate systems**
  • Each pixel has a coordinate
  • Draw something at a pixel by
    • Specifying what to draw
    • Specifying where to draw

• Do we draw each pixel?
  • Use a **drawing API**
  • Given an image; does work
  • This is what XNA provides
Sprite Coordinate Systems

- **Screen coordinates**: where to paint the image
  - Think screen pixels as a coordinate system
  - Very important for object *transformations*
    - **Example**: scale, rotate, translate
  - In 2D, origin is typically top left of screen

- **Object coordinate**: location of pixels in object
  - Think of sprite as an image file (it often is)
  - Coordinates are location of pixels in this file
  - Unchanged when object moves about screen
Sprite Coordinate Systems

Object: (0,0)

Screen: (300,200)
Drawing Sprites

• **Basic instructions:**
  - Set origin for the image in *object coordinates*
  - Give the `SpriteBatch` a point to draw at
  - Screen places origin of image at that point

• What about the other pixels?
  - Depends on transformations (rotated? scaled?)
  - But these (almost) never affect the origin

• Sometimes we can **reset** the object origin
Sprite Coordinate Systems

Object: (0,0)

Screen: (300,200)
Sprite Coordinate Systems

(0,0)  +x

Screen: (300,200)  Object: (0,0)

+y
Sprite Coordinate Systems

(0,0) +x

Screen: (300,200)  
Object: (0,50)

2D Sprite Graphics
Drawing with SpriteBatch

```
Draw(GameTime time) {
    ...
    spriteBatch.Begin();
    spriteBatch.Draw(image0,origin0,color0);
    spriteBatch.Draw(image1,origin1,color1);
    ...
    spriteBatch.End();
    ...
}
```
2D Transforms

• A function $T : \mathbb{R}^2 \rightarrow \mathbb{R}^2$
  • “Moves” one set of points to another set of points
  • Transforms one “coordinate system” to another
  • The new coordinate system is the distortion

• Idea: Draw on paper and then “distort” it
  • Examples: Stretching, rotating, reflecting
  • Determines placement of “other” pixels
  • Also allows us to get multiple images for free
The “Drawing Transform”

- $T : \text{object coords} \rightarrow \text{screen coords}$
  - Assume pixel $(a,b)$ in art file is blue
  - Then screen pixel $T(a,b)$ is blue
  - We call $T$ the object map

- By default, object space = screen space
  - Color of image at $(a,b)$ = color of screen at $(a,b)$
  - By drawing an image, you are transforming it

- $S$ an image; transformed image is $T(S)$
Example: Translation

- Simplest transformation: \( T(v) = v + u \)
  - Shifts object in direction \( u \)
  - Distance shifted is magnitude of \( u \)

- Used to place objects on screen
  - By default, object origin is screen origin
  - \( T(v) = v + u \) places object origin at \( u \)
Composing Transformations

- **Example:** \( T : \mathbb{R}^2 \rightarrow \mathbb{R}^2, \ S : \mathbb{R}^2 \rightarrow \mathbb{R}^2 \)
  - Assume pixel \((a,b)\) in art file is blue
  - Transform \( T \) makes pixel \( T(a,b) \) blue
  - Transform \( S \circ T \) makes pixel \( S(T(a,b)) \) blue

- **Strategy:** use transforms as building blocks
  - Think about what you want to do visually
  - Break it into a sequence of transforms
  - Compose the transforms together
Application: Scrolling

World origin

Screen origin

Object origin

World

Screen

2D Sprite Graphics
Scrolling: Two Translations

- Place object in the World at point $p = (x, y)$
  - Basic drawing transform is $T(v) = v + p$

- Suppose Screen origin is at $q = (x', y')$
  - Then object is on the Screen at point $p - q$
  - $S(v) = v - q$ transforms World coords to Screen
  - $S \circ T(v)$ transforms the Object to the Screen

- This separation makes scrolling easy
  - To move the object, change $T$ but leave $S$ same
  - To scroll the screen, change $S$ but leave $T$ same
Scrolling: Practical Concerns

- Many objects will exist outside the screen
  - Can draw if want; graphics card will drop them
  - It is expensive to keep track of them all
  - But is also unrealistic to always ignore them

- In graphics, drawing transform = matrix
  - Hence composition = matrix multiplication
  - Details beyond the scope of this course
  - XNA handles all of this for you (sort of)
Using Transforms in XNA

- XNA has methods for creating transforms
  - Important: transforms are all 3D, not 2D
    - Just make sure the $z$-value is always 0
  - Methods are part of the `Matrix` class
    - `Matrix.CreateTranslation(x, y, z)`

- Parameters fill in details about transform
  - Example: Position ($x, y, z$) if a translation
  - The most math you will ever need for this
Positioning in XNA

Draw(GameTime time) {

    Vector2 oPos = object.Position();

    spriteBatch.Begin();
    spriteBatch.Draw(image, oPos, color);
    spriteBatch.End();
}

2D Sprite Graphics
Positioning in XNA

Draw(GameTime time) {
    Vector2 origin = new Vector2(0,0);
    Vector2 oPos = object.Position();
    Matrix tran = Matrix.CreateTranslation(oPos.x,oPos.y,0);
    spriteBatch.Begin(sort,blend,null,null,null,tran);
    spriteBatch.Draw(image,origin,color);
    spriteBatch.End();
}

Translate origin to position in world.

too advanced for this class

2D Sprite Graphics
Positioning in XNA

```csharp
Draw(GameTime time) {
    Vector2 origin = new Vector2(0,0);
    Vector2 oPos = object.Position();
    Matrix tran = Matrix.CreateTranslation(oPos.x,oPos.y,0);
    Vector2 wPos = viewWindow.Position();
    Matrix wTran = Matrix.CreateTranslation(-wPos.x,-wPos.y,0);
    Matrix tran = Matrix.multiply(wTran,oTran);
    spriteBatch.Begin(sort,blend,null,null,null,null,null,tran);
        spriteBatch.Draw(image,origin,color);
    spriteBatch.End();
}
```
Positioning in XNA

```csharp
Draw(GameTime time) {
    Vector2 origin = new Vector2(0,0);
    Vector2 oPos = object.Position();
    Matrix tran = Matrix.CreateTranslation(oPos.x, oPos.y, 0);
    Vector2 wPos = viewWindow.Position();
    Matrix wTran = Matrix.CreateTranslation(-wPos.x,-wPos.y,0);
    Matrix tran2 = Matrix.CreateTranslation(0,0,0);
    Matrix tran3 = Matrix.multiply(wTran, tran2);
    spriteBatch.Begin(sort, blend, null, null, null, null, tran3);
    spriteBatch.Draw(image, origin, color);
    spriteBatch.End();
}
```

2D Sprite Graphics
A Hybrid Approach

Draw(GameTime time) {
    Vector2 oPos = object.Position();
    // Just get translation for window
    Vector2 wPos = viewWindow.Position();
    Matrix tran = Matrix.CreateTranslation(-wPos.x,-wPos.y,0);
    // Apply window translation to contents of SpriteBatch
    spriteBatch.Begin(sort,blend,null,null,null,null,null,tran);
    // Use oPos for origin
    spriteBatch.Draw(image,oPos,color);
    spriteBatch.End();
}
An Alternative: Effect Passes

Draw(GameTime time) {
    BasicEffect effect;
    effect.world = tran1;       // Store the first transform
    spriteBatch.Begin();
        effect.CurrentTechnique.Passes[0].Apply();
    spriteBatch.Draw(image1, origin, color);
    effect.world = tran2;       // Store the second transform
    spriteBatch.Draw(image2, origin, color);
    spriteBatch.End();
}

2D Sprite Graphics
Matrix Transform Gallery

Uniform Scale: 
\[
\begin{bmatrix}
  s & 0 \\
  0 & s \\
\end{bmatrix}
\begin{bmatrix}
  x \\
  y \\
\end{bmatrix} =
\begin{bmatrix}
  sx \\
  sy \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
  1.5 & 0 \\
  0 & 1.5 \\
\end{bmatrix}
\]

Matrix.CreateScale(s);
Matrix Transform Gallery

- Nonuniform Scale:

\[
\begin{bmatrix}
  s_x & 0 \\
  0 & s_y
\end{bmatrix}
\begin{bmatrix}
  x \\
  y
\end{bmatrix}
= 
\begin{bmatrix}
  s_x x \\
  s_y y
\end{bmatrix}
\]

\[
\begin{bmatrix}
  1.5 & 0 \\
  0 & 0.8
\end{bmatrix}
\]

Matrix.CreateScale(sx,sy,1);
Matrix Transform Gallery

Rotation:

\[
\begin{bmatrix}
\cos \theta & -\sin \theta \\
\sin \theta & \cos \theta \\
0.866 & -0.5 \\
0.5 & 0.866
\end{bmatrix}
\begin{bmatrix}
x \\
y
\end{bmatrix}
= 
\begin{bmatrix}
x \cos \theta - y \sin \theta \\
x \sin \theta + y \cos \theta
\end{bmatrix}
\]

\[\text{Matrix.CreateRotationZ(angle);}\]
Matrix Transform Gallery

- Reflection: \[
\begin{bmatrix}
-1 & 0 \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y
\end{bmatrix}
= \begin{bmatrix}
-x \\
y
\end{bmatrix}
\]

- View as special case of Scale
  \[
\begin{bmatrix}
-1 & 0 \\
0 & 1
\end{bmatrix}
\]
Matrix Transform Gallery

- Shear:

\[
\begin{bmatrix}
1 & a \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y
\end{bmatrix} =
\begin{bmatrix}
x + ay \\
y
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 0.5 \\
0 & 1
\end{bmatrix}
\]
Image-By-Image Transforms

spriteBatch.Draw(
    image,  // What want to draw
    size,   // Rectangle to fit image inside
    crop,   // Rectangle to crop image
    color,  
    rot,    // Amount to rotate image
    pos,    // Position to place image
    ref,    // How to reflect the image
    depth   // How to layer image
);

2D Sprite Graphics
Image-By-Image Transforms

spriteBatch.Draw(
    image, // What want to draw
    size,  // Rectangle to fit image
    crop,  // Rectangle to crop
    color, // How to color
    rot,   // Amount to rotate image
    pos,   // Position to place image
    ref,   // How to reflect the image
depth); // How to layer image

Performs them in the order given
Compositing Transforms

- In general not commutative: order matters!

rotate, then translate

translate, then rotate
Compositing Transforms

• In general not commutative: order matters!

scale, then rotate

rotate, then scale
A Word About Scaling

- If making smaller, it drops out pixels
  - Suppose $T(v) = 0.5v$
  - (0,0) = $T(0,0)$; pixel (0,0) colored from (0,0) in file
  - (0,1) = $T(0,2)$; pixel (0,1) colored from (0,2) in file

- But if making larger, it duplicates pixels
  - Suppose $T(v) = 2v$
  - (0,1) = $T(0,0.5)$; pixel (0,1) colored from (0,1) in file
  - (0,1) = $T(0,1)$; pixel (0,2) colored from (0,1) in file

- This can lead to *jaggies*
Scaling and Jaggies

- **Jaggies**: Image is blocky
- Possible to smooth image
  - Done through blurring
  - In *addition* to transform
  - *Some* graphic card support
- Solution for games
  - Shrinking is okay
  - Enlarging not (always) okay
  - Make sprite large as needed
Summary

- Drawing is all about coordinate systems
  - **Object coords**: Coordinates of pixels in image file
  - **Screen coords**: Coordinates of screen pixels

- Transforms alter coordinate systems
  - “Multiply” image by matrix to distort them
  - Multiply transforms together to combine them
    - Matrices are not commutative
    - First transform goes on “the right”