Lecture 14

2D Sprite Graphics
Graphics Lectures

- Drawing Images
  - SpriteBatch interface
  - Coordinates and Transforms

- Drawing Perspective
  - Camera
  - Projections

- Drawing Primitives
  - Color and Textures
  - Polygons
Graphics Lectures

• Drawing Images
  • SpriteBatch interface
  • Coordinates and Transforms

• Drawing Perspective
  • Camera
  • Projections

• Drawing Primitives
  • Color and Textures
  • Polygons

bare minimum to draw graphics
side-scroller vs. top down
necessary for lighting & shadows
Graphics Lectures

- Drawing Images
  - SpriteBatch interface
  - Coordinates and Transforms
- Drawing Perspective
  - Camera
  - Projections
- Drawing Primitives
  - Color and Textures
  - Polygons

Animation is part of AI Lectures
Graphics Lectures

- **Drawing Images**
  - SpriteBatch interface
  - Coordinates and Transforms
  - bare minimum to draw graphics

- **Drawing Perspective**
  - Camera
  - Projections
  - side-scroller vs. top down

- **Drawing Primitives**
  - Color and Textures
  - Polygons
  - 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 LibGDX, use the class 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
  - What LibGDX gives us
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, LibGDX origin is **bottom 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
Screen: (300, 200)
Object: (0, 0)
Historical Coordinate Systems

Screen: (300, 200)  Object: (0, 0)
Historical Coordinate Systems

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

Mouse coordinates still do this (see Loading.java in labs)
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

Screen: (300,200)
Object: (0,0)
Sprite Coordinate Systems

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

Sprite Coordinate Systems

Screen: (300, 200)
Object: (0, 0)
Sprite Coordinate Systems

Screen: (300,200)
Object: (0,0)
Sprite Coordinate Systems

Screen: (300,200)
Object: (0,0)
public void draw(float dt) {

    spriteBatch.begin();
spriteBatch.draw(image0);
spriteBatch.draw(image1, pos.x, pos.y);

    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 \textbf{object map}

- By default, object space = screen space
  - Color of image at $(a,b) = \text{color of screen at } (a,b)$
  - By drawing an image, you are \textit{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 Transforms

- **Example**: $T : \mathbb{R}^2 \to \mathbb{R}^2$, $S : \mathbb{R}^2 \to \mathbb{R}^2$
  - 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
Application: Scrolling
Application: Scrolling

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 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
  - LibGDX handles all of this for you (sort of)
Using Transforms in LibGDX

- LibGDX has methods for creating transforms
  - Two types depending on application
    - Affine2 for transforming 2D sprites
    - Matrix4 for transforming 3D object
      - But also for transforming fonts
  - Parameters fill in details about transform
    - Example: Position \((x,y)\) if a translation
    - The most math you will ever need for this
Transforms in SpriteBatch

**Affine2**
- Pass it to a draw command
  - Applies only to that image
  - Adds to CPU power
- Handles everything
  - Location is in transform
  - Transform to object position
- `sb.draw(image, wd, ht, affine);`

**Matrix4**
- Pass to `setTransformMatrix`
  - Applies to all images!
  - Handled by the GPU but…
  - Change causes GPU stall
- Only use this if you must
  - e.g. Transforming fonts
  - See `GameCanvas` in Lab1
Transforms in SpriteBatch

**Affine2**
- Pass it to a draw command
  - Applies only to that image
  - Adds to CPU power
- Handles everything
  - Location is in transform
  - Transform to object position
- `sb.draw(image, wd, ht, affine);`

**Matrix4**
- Pass to setTransformMatrix
  - Applies to all images!
  - Handled by the GPU but…
  - Change causes GPU stall
- Only use this if you must
  - e.g. Transforming fonts
  - See GameCanvas in Lab1

Only supports a **TextureRegion**??
public void draw(float dt) {

    Vector2 pos = object.getPosition();

    spriteBatch.begin();
    spriteBatch.draw(image, pos.x, pos.y);
    spriteBatch.end();
}

2D Sprite Graphics
public void draw(float dt) {
    Affine2 oTran = new Affine2();
oTran.setToTranslation(object.getPosition());
spriteBatch.begin();
spriteBatch.draw(image, width, height, oTran);
spriteBatch.end();
}
public void draw(float dt) {
    Affine2 oTran = new Affine2();
    oTran.setToTranslation(object.getPosition());
    Affine2 wtran = new Affine2();
    Vector2 wPos = viewWindow.getPosition();
    wTran.setToTranslation(-wPos.x,-wPos.y);
    oTran.mul(wTran);
    spriteBatch.begin();
    spriteBatch.draw(image,width,height,oTran);
    spriteBatch.end();
}
Transform Gallery

- Uniform Scale:

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

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

`affine.setToScaling(s,s);`
Uniform Scale: 

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

Represent as 2x2 matrix

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

\text{affine.setToScaling}(s, s);
Matrix Transform Gallery

- Nonuniform Scale:

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

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

affine.setToScaling(sx, sy);
Matrix Transform Gallery

- Rotation:

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

affine.setToRotationRad(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:**

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

\[
\begin{bmatrix}
1 & 0.5 \\
0 & 1 \\
\end{bmatrix}
\]

```javascript
affine.setToShearing(a,1);
```
Translation Revisited

- Translation is **not** a linear transform
  - To be linear, \( T(v+w) = T(v)+T(w) \)
  - Translation transform is \( T(v) = v+u \)
  - \( T(v)+T(w) = (v+u)+(w+u) = v+w+2u \neq T(v+w) \)

- But LibGDX treats it like one
  - **Affine2** transforms support translation
  - **Matrix4** supports `matrix.set(affine)`

- What is going on here?
Homogenous Coordinates

- Add an **extra dimension** to the calculation.
  - An extra component $w$ for vectors
  - For affine transformations, can keep $w = 1$
  - Add extra row, column to matrices (so $3 \times 3$)

- Dimension is for calculation only
  - We are not in 3D-space **yet**
  - 3D transforms need 4D vectors, $4 \times 4$ matrices

- Matrix4 because LibGDX supports 3D

2D Sprite Graphics
Homogenous Coordinates

- Linear transforms have dummy row and column

\[
\begin{bmatrix}
a & b & 0 \\
c & d & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
1
\end{bmatrix}
= 
\begin{bmatrix}
ax + by \\
cx + dy \\
1
\end{bmatrix}
\]

- Translation uses extra column

\[
\begin{bmatrix}
1 & 0 & t \\
0 & 1 & s \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
1
\end{bmatrix}
= 
\begin{bmatrix}
x + t \\
y + s \\
1
\end{bmatrix}
\]

2D Sprite Graphics
Affine Transforms Revisited

- **Affine**: Linear on homogenous coords
  - Equal to all transforms $T(v) = Mv + p$
  - Treat everything as matrix multiplication

- Why does this work?
  - Area of mathematics called projective geometry
  - Far beyond the scope of this class

- LibGDX hides all the messy details
  - Just stick with Affine2 class for now
Affine Transform Gallery

- Translation:

\[
\begin{bmatrix}
1 & 0 & t_x \\
0 & 1 & t_y \\
0 & 0 & 1
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 0 & 2.15 \\
0 & 1 & 0.85 \\
0 & 0 & 1
\end{bmatrix}
\]

2D Sprite Graphics
Affine Transform Gallery

- **Uniform Scale:**

\[
\begin{bmatrix}
  s & 0 & 0 \\
  0 & s & 0 \\
  0 & 0 & 1
\end{bmatrix}
\quad \begin{bmatrix}
  1.5 & 0 & 0 \\
  0 & 1.5 & 0 \\
  0 & 0 & 1
\end{bmatrix}
\]
Affine Transform Gallery

- Nonuniform Scale:

\[
\begin{bmatrix}
    s_x & 0 & 0 \\
    0 & s_y & 0 \\
    0 & 0 & 1 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
    1.5 & 0 & 0 \\
    0 & 0.8 & 0 \\
    0 & 0 & 1 \\
\end{bmatrix}
\]

2D Sprite Graphics
Affine Transform Gallery

- Rotation:

\[
\begin{bmatrix}
\cos \theta & -\sin \theta & 0 \\
\sin \theta & \cos \theta & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
0.866 & -0.5 & 0 \\
0.5 & 0.866 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

2D Sprite Graphics
Affine Transform Gallery

- Reflection:
  - Special case of Scale

\[
\begin{bmatrix}
-1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]
Affine Transform Gallery

• Shear:

\[
\begin{bmatrix}
1 & a & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 0.5 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

2D Sprite Graphics
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
Rotating Object About Center

- Translate center to origin
- Rotate about origin
- Translate to object position
Rotating Object About Center

- Translate center to origin
- Rotate about origin
- Translate to object position
Rotating Object About Center

- Translate center to origin
- Rotate about origin
- Translate to object position
Rotating Object About Center

- Translate center to origin
- Rotate about origin
- Translate to final position
Rotating Object About Center

- Translate center to origin
- Rotate about origin
- Translate to final position
Transforms and Modular Animation

- Break asset into parts
  - Natural for joints/bodies
  - Animate each separately
- Cuts down on filmstrips
  - Most steps are transforms
  - A lot less for you to draw
  - Also better for physics
- Several tools to help you
  - **Example:** Spriter, Spine
  - Great for visualizing design
Transforms and Modular Animation

• Break asset into parts
  • Natural for joints/bodies
  • Animate each separately

• Cuts down on filmstrips
  • Most steps are transforms
  • A lot less for you to draw
  • Also better for physics

• Several tools to help you
  • Example: Spriter, Spine
  • Great for visualizing design
Transforms and Modular Animation

- Break asset into parts
  - Natural for joints/bodies
  - Animate each separately
- Cuts down on filmstrips
  - Most steps are transforms
  - A lot less for you to draw
  - Also better for physics
- Several tools to help you
  - **Example**: Spriter, Spine
  - Great for visualizing design
Spine Demo
**Spine Demo**

More on this in AI Lecture
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
  • Later transforms go on “the right”