Sprite Graphics

CS 4730 – Computer Game Design

Slide material courtesy Walker White (Cornell)

First, a history lesson…

- We start with the CRT (cathode ray tube)
  - Made up of pixels (picture elements)
  - A 300x200 screen would have 200 scan lines of 300 pixels each
  - Each line would get drawn, starting from the top and working it's way down
  - This is one reason graphics coordinates start with the upper left

History

- After drawing all the scan lines (aka one frame), the ray gun had to line back up at the top
- This delay is called VBLANK
  - It's what makes a PAL TV different than an NTSC, etc.
- Early game systems didn't have enough memory to store one screen's worth of pixels at one time!

Synctastic

- If a game (old or modern) updates the frame buffer when the scan lines are in the middle of being drawn, tearing occurs
- One option is to lock the update() method to VBLANK (not the best option)
- Another is to have more than one buffer and swap the two during VBLANK
- This swapping is called VSYNC
- Good frame rates lock to VSYNC

Pretty, Pretty Pictures

- "Though still rooted in the work of the first two games, X improves on the visuals even more and features impressive graphical effects."
- "And despite all of their saccharin-sweet cuteness, the graphics are magnificent. X's dream of producing an interactive cartoon has been fully realized -- the animation is lavish, the textures rich, and even the most superfluous touches have been completely executed."

Pretty, Pretty Pictures

- Castlevania 3 (NES)
- Super Mario 64 (N64)

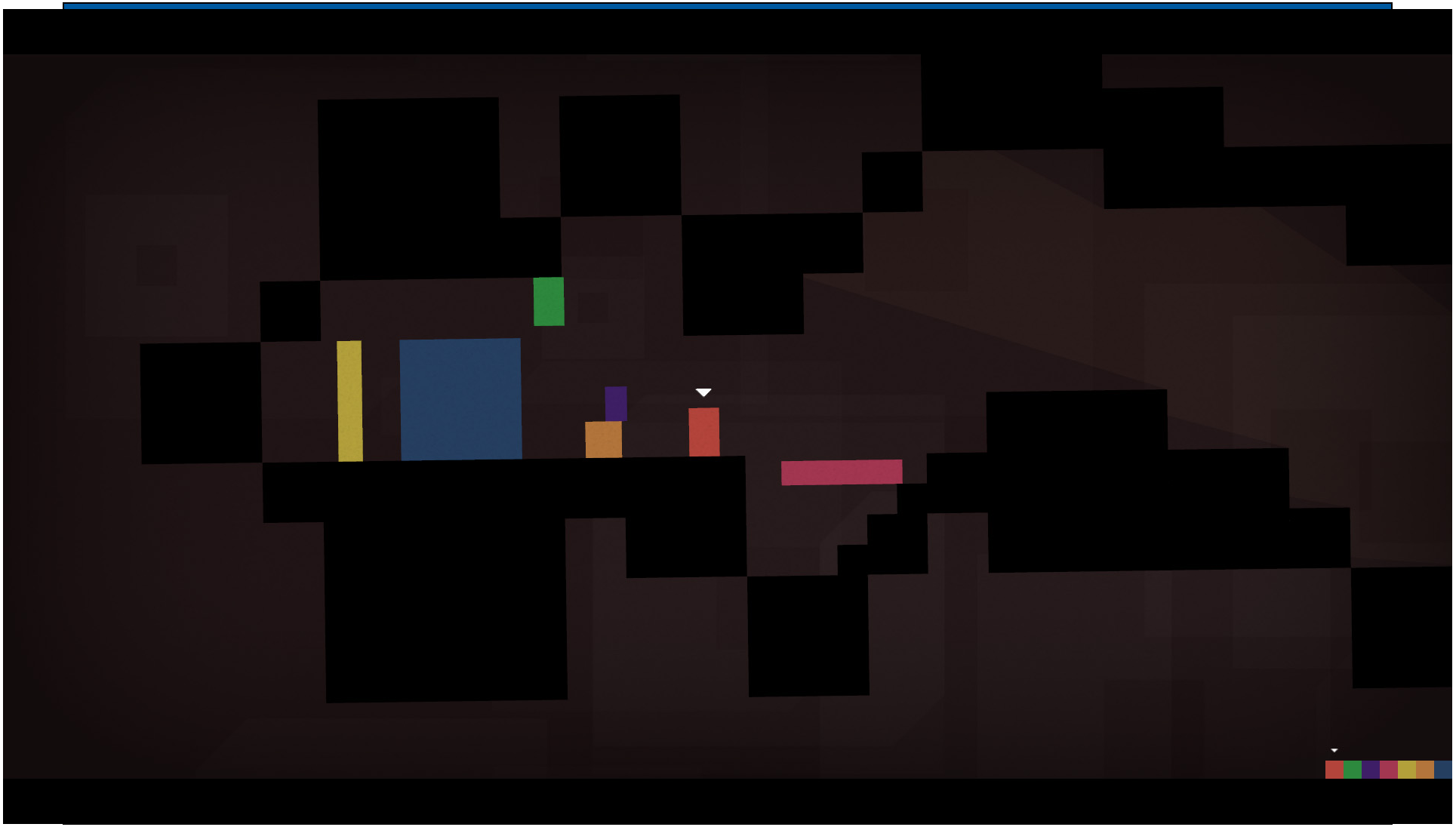
- So… it's all relative, huh?

Graphics vs. Visual Design

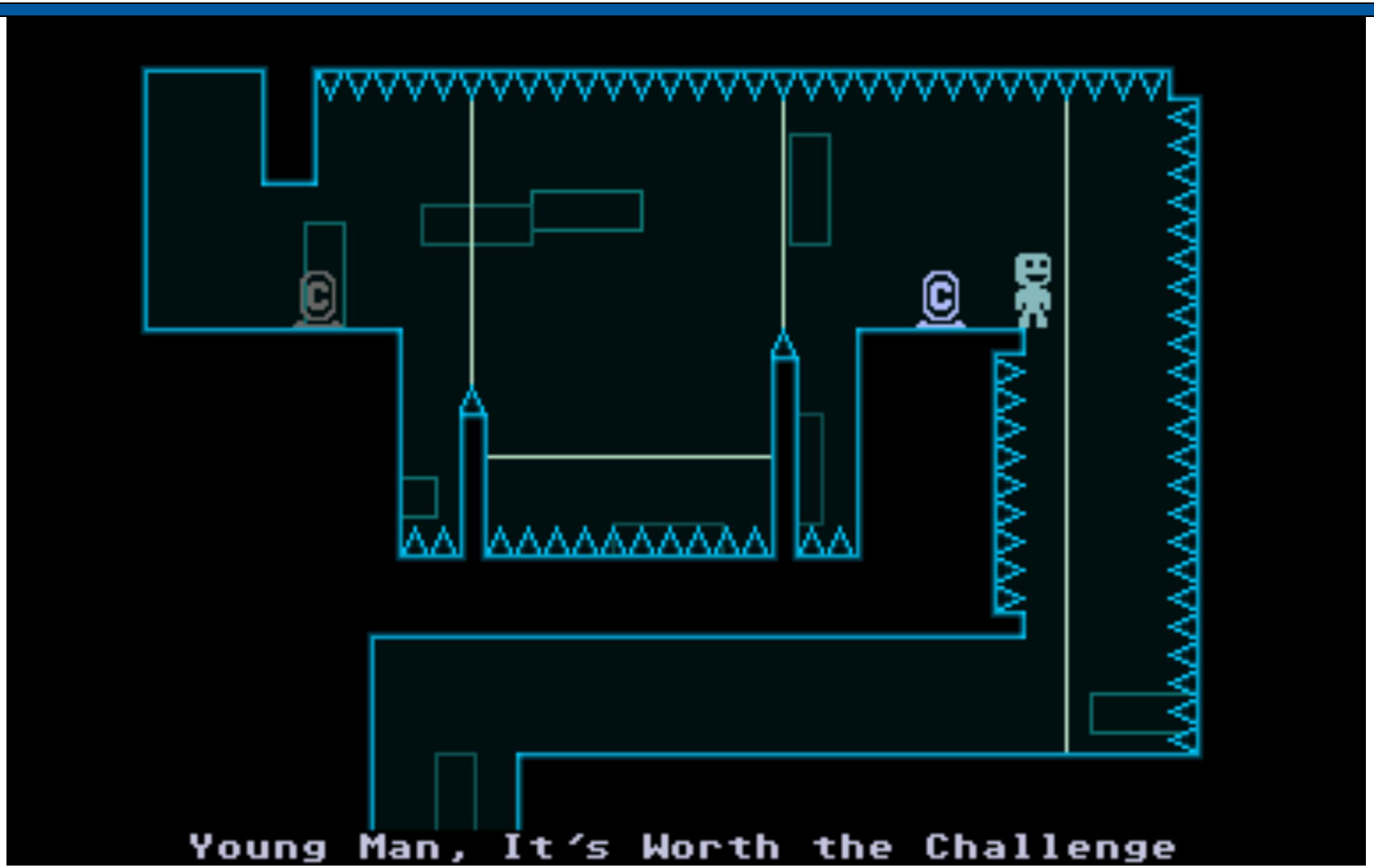
- Sometimes (most times) you don't have to have AMAZING graphics
- You need to have the RIGHT graphics
- Tetris wouldn't be "better" with sweet bump mapping
- Just because we're using 2D, that doesn't mean you're being held back

Graphics vs. Visual Design

Graphics vs. Visual Design



Graphics vs. Visual Design



Graphics vs. Visual Design

The SpriteBatch

- The SpriteBatch is exactly what it sounds like:
  - A way to draw a bunch of Sprites all at once (aka "in a batch")
  - SpriteBatch composites Sprites on top of each other, like cels of animation
- With each frame, we need to clear the screen, stack the sprites, then draw them all to the screen (and update the system that we have drawn)

The SpriteBatch

```
Texture2D background = Content.Load<Texture2D>("stars");
Texture2D shuttle = Content.Load<Texture2D>("shuttle");
Texture2D earth = Content.Load<Texture2D>("earth");

protected override void Draw(GameTime gameTime)
{
    graphics.GraphicsDevice.Clear(Color.CornflowerBlue);
    spriteBatch.Begin();
    spriteBatch.Draw(background, new Rectangle(0, 0, 800, 480),
Color.White);
    spriteBatch.Draw(earth, new Vector2(400, 240), Color.White);
    spriteBatch.Draw(shuttle, new Vector2(450, 240), Color.White);
    spriteBatch.End();
    base.Draw(gameTime);
}
```

The SpriteBatch

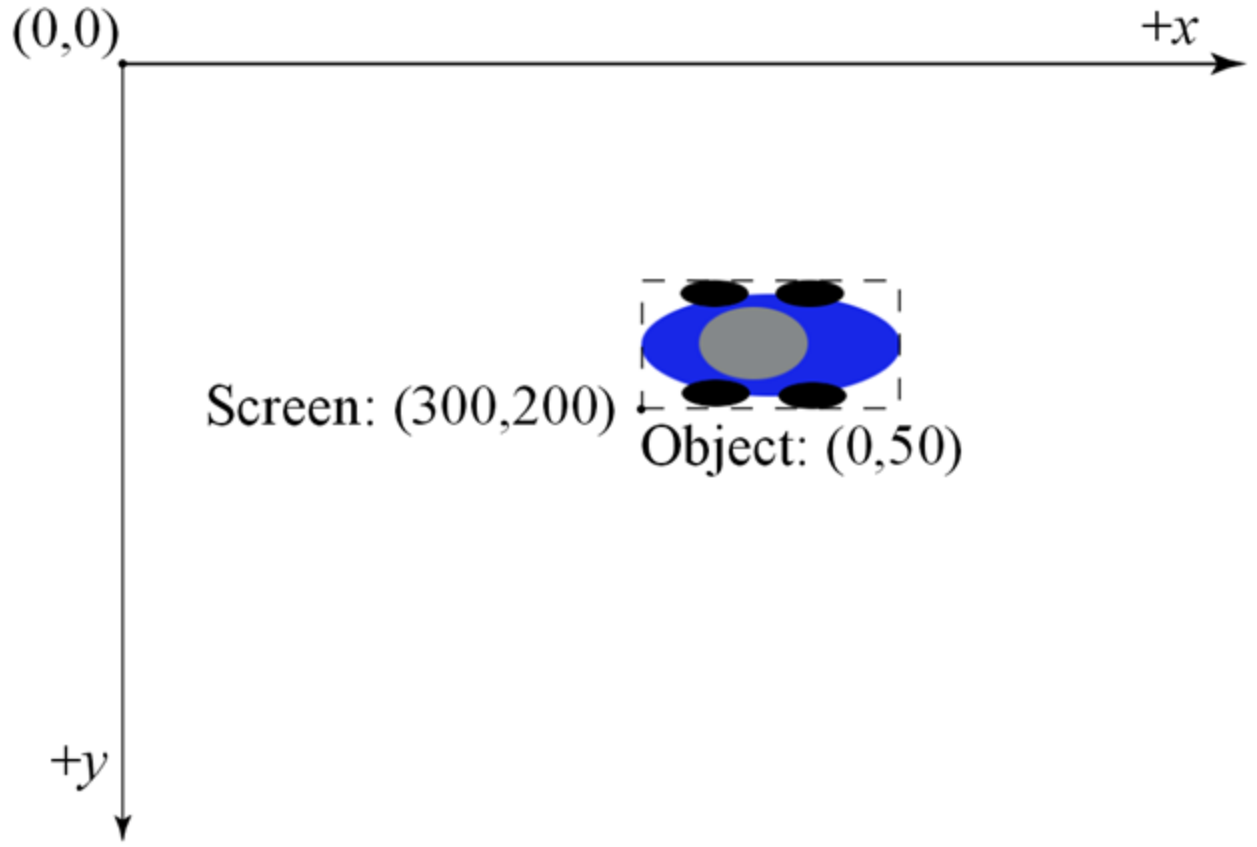
- What this gives us is the library methods we need to know whether we HAVE to draw each pixel
- If two images overlap, which color do you make the pixel? This is handled for us!

Sprite Coordinates

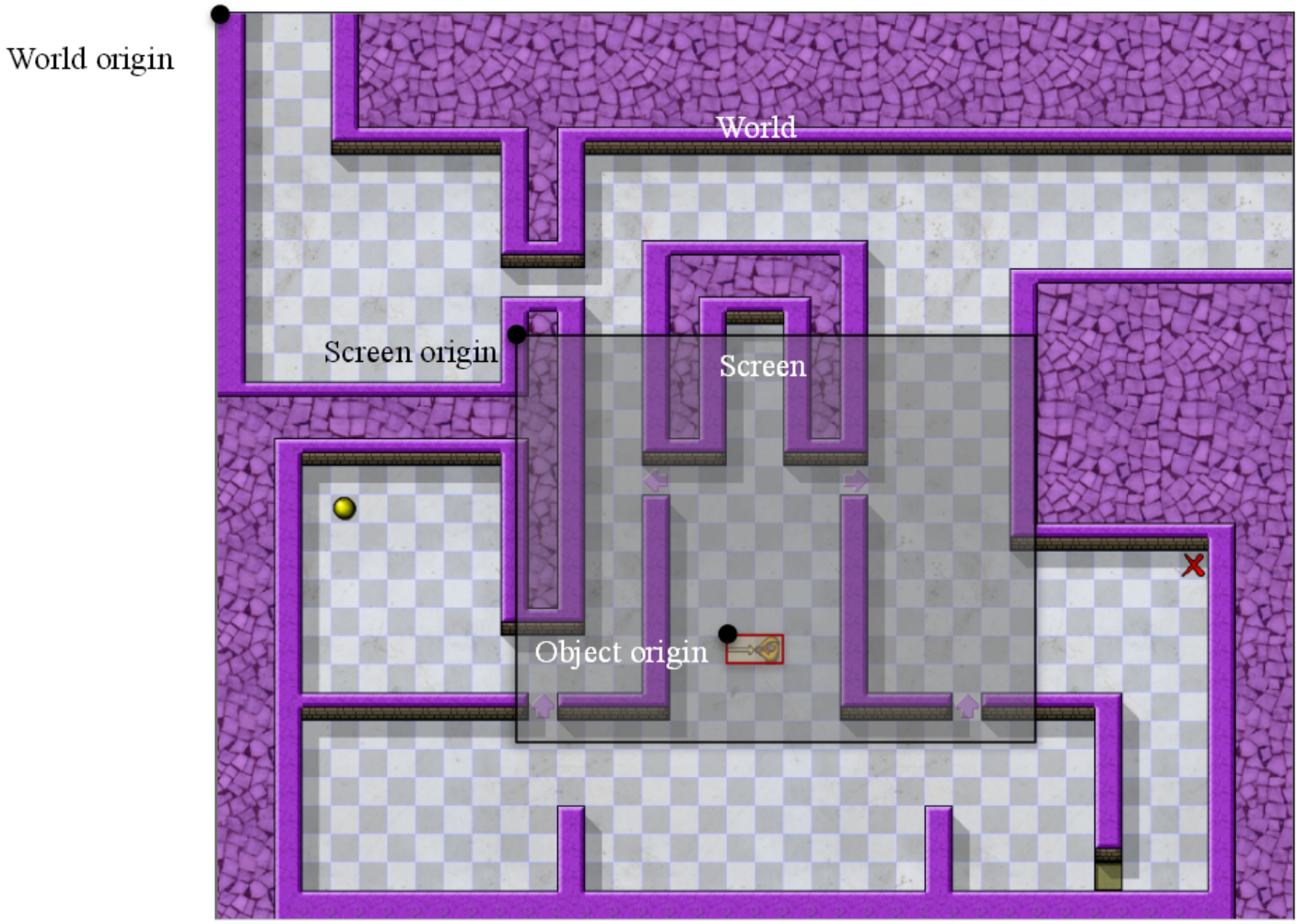
- Sprites are an abstraction of all graphical content
- Three kinds of coordinates: world, screen, and object
- We can manipulate sprites using linear algebra
  - Some examples include translation, scaling, and rotation

Object coordinates

- Position of a pixel within an object



Screen and world coordinates



Drawing Back to Front

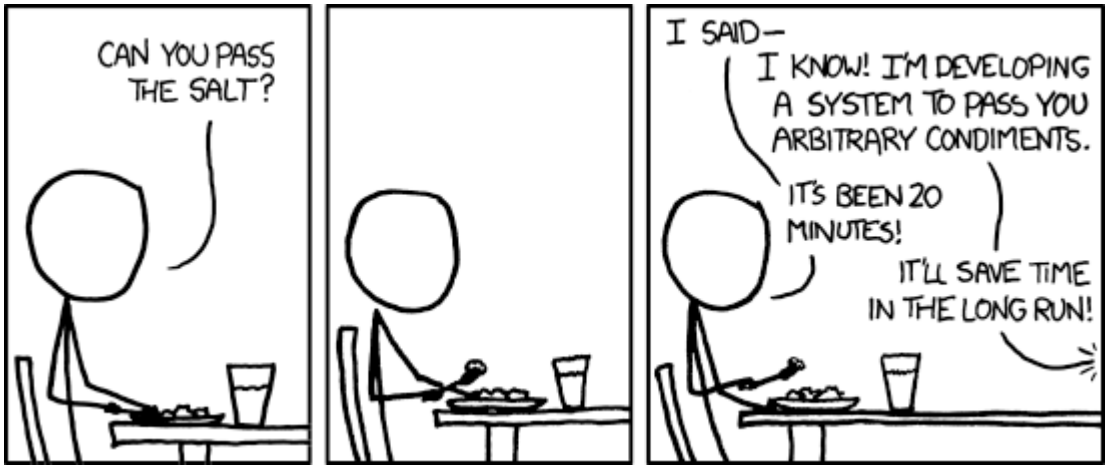
- End() is easy – you call it when you're finished drawing things!
- Draw() is a touch more complex, but still simple. Three arguments:
  - The image to draw
  - The position at which to draw
  - A color to tint it (use Color.White most of the time)

e.g.

```
mySpriteBatch.Draw(background, Vector2.Zero, Color.White);
```

Back to front (continued)

- The Begin() method is the most complex
- You can pass it a variety of parameters that control how everything is drawn
- Or you can just pass it nothing (easiest and therefore most recommended choice)



2D Transformations

- A function *T :*
  - Moves one set of points to another set of points
  - Transforms one coordinate system to another
  - The new coordinate system is the distortion
- The Idea: Draw then "distort" it
  - Examples: Stretching, rotating, reflecting
  - Allows us to get multiple images for free! (i.e. character facing both directions)

The Drawing Transform

- The simplest transformation is a direct mapping
  - Assume pixel *(a,b)* of the image is blue
  - Then screen pixel *T(a,b)* is blue
- By default, the object space = screen space
  - The color at one point in the image = the color of the screen's point
  - The draw is the actual transformation

Translation Transformation

- $T(v) = v + u$
  - Here, we shift an object in the direction of u
  - Distance shifted is the magnitude of u
- This is what is used to place objects on the screen
- By default, object origin is screen origin
- This transformation places the object origin at u

Composing Transformations

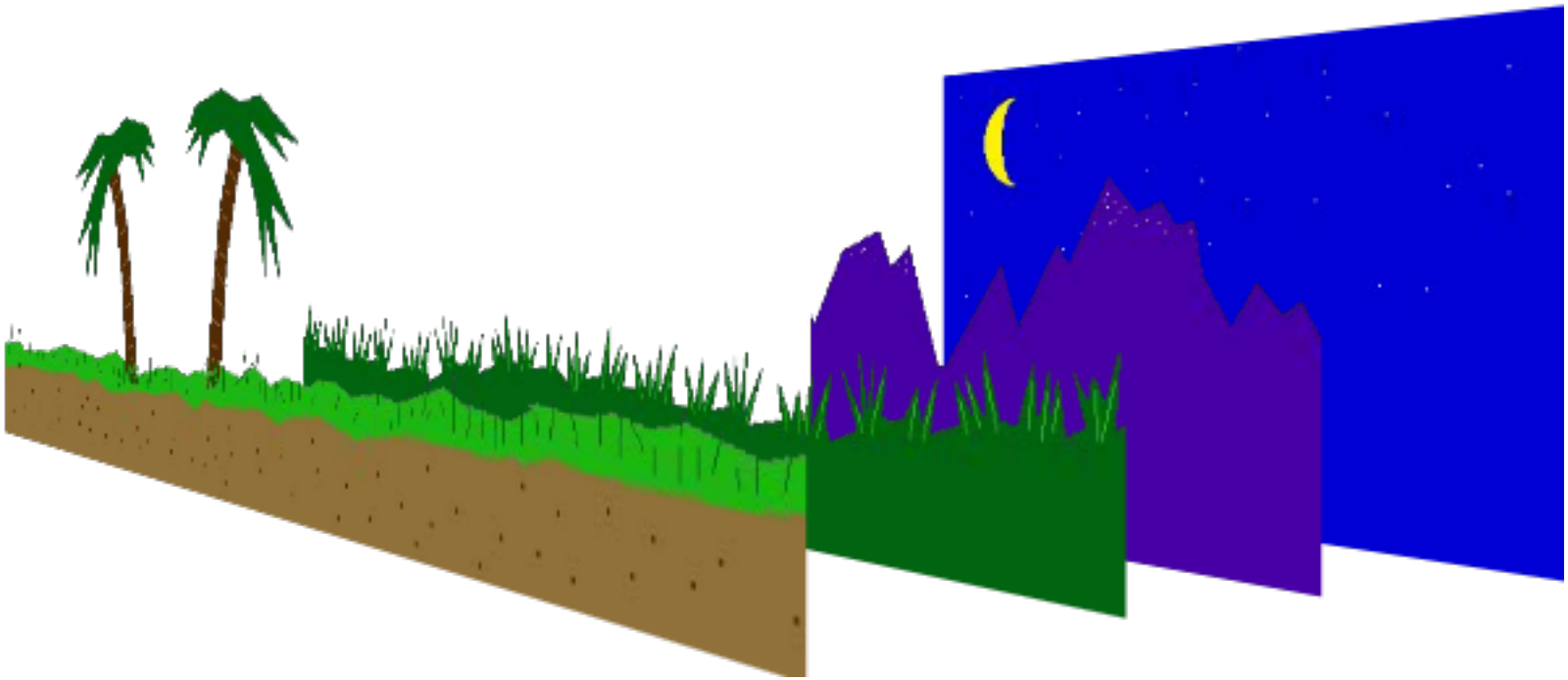
- Assume we have two transformations, *T* and *S*
  - Assume pixel *(a,b)* is blue in the image
  - Transform *T* makes *T(a,b)* blue
  - Transform ST makes *S(T(a,b))* blue
- Consider what you want the image to do, then break it down into a set of transformations

Scrolling

- Place object in the World at p = *(x,y)*
  - i.e. *T(v) = v+p*
- Suppose the screen origin is at q = *(x', y')*
  - Then the object on the screen is at point p-q
  - *S(v) = v-q* transforms World coords to Screen
  - *ST(v)* transforms Object to the Screen
- Thus to scroll:
  - To move the object, change *T,* but not *S*
  - To scroll the screen, change *S*, but not *T*

Parallax Scrolling

- Multiple layers of background, scrolling at different speeds to give the illusion of depth



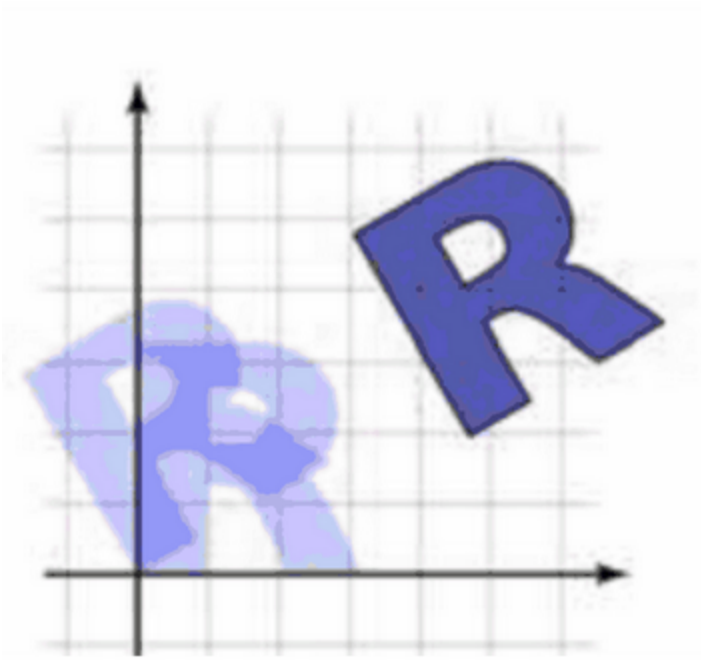
Scrolling

- What about objects not on screen?
  - If you try to draw them, that's fine
  - The graphics card will just ignore them
- But what does that do to your game loop?
  - Keeping up with all those objects…
  - Updating all those locations…

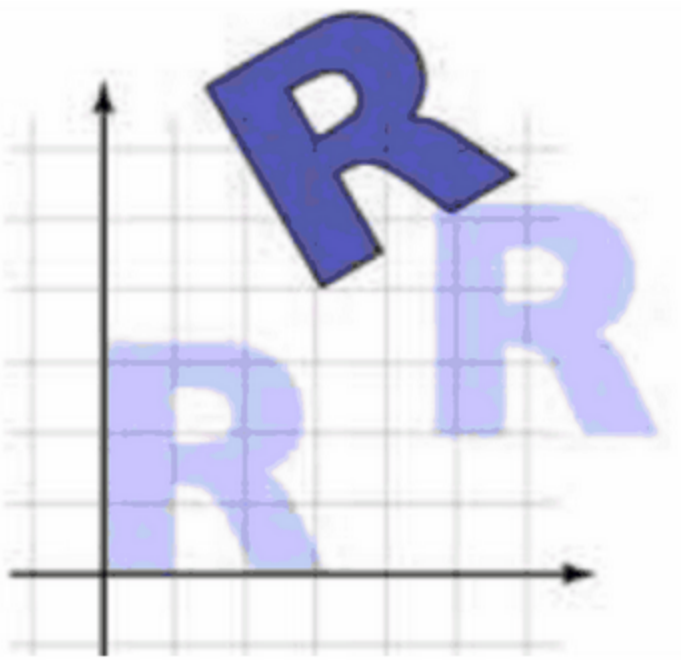
Beware Some Transforms!

- Scaling can be tough
- If you up-scale an image, you are increasing the number of pixels, but not the amount of information
- This leads to “jaggies” or blocky images
- Scaling down is okay (usually)

Beware Transform Order!



Rotate then Translate



Translate then Rotate

Animation

- Not quite as simple as you might hope…
- Roll your own AnimatedSprite class
  - Here you would load the sprite sheet and keep up with which frame is up next
- Try a third-party tool like TexturePacker

Sprite Sheets

What makes a game 3D?

- For the most part, the game is still rendered in 2D for a flat monitor screen…
- So how is it 3D?

What makes a game 3D?

- Two main aspects:
  - Assets are rendered on all sides
  - User has some extra control of the camera to move in all three directions

The Camera

- When we refer to the camera in a game, we are referring to the angle and perspective of the player
- Represented as it's own coordinate system
- Moving the camera is another coordinate transformation!

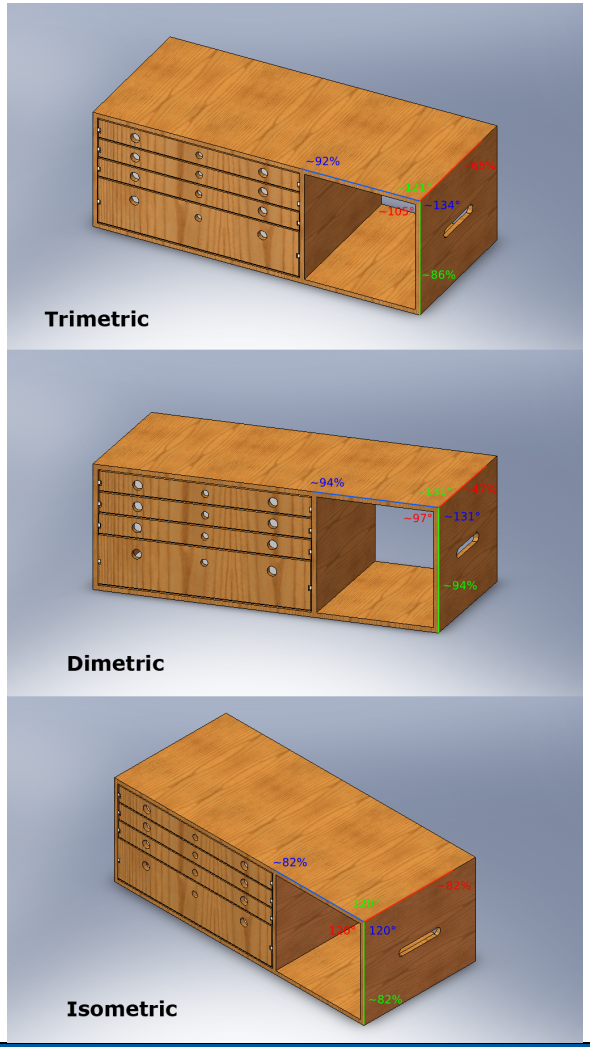
Some Types of Cameras

- Orthographic
  - perpendicular to the plane of the game
  - Z-axis? Top-down perspective.
  - Y-axis? Side scroller.
- Handy for artists!
  - Art is done as tiles and easy to manage
- Forces 2D gameplay only

Some Types of Cameras

- Axonometric
  - "off axis" view
  - Allows for some 3D gameplay
  - Isometric is most common form
    - All axes are equal
  - Can have other forms by shortening/lengthening some axes for artistic reasons

Some Types of Cameras



Some Types of Cameras

Some Types of Cameras

Some Types of Cameras

Some Types of Cameras

Some Types of Cameras

Some Types of Cameras

Some Types of Cameras