2D Graphics Basics

Sean Middleditch - DigiPen

Game Engine Architecture

About Yours Truly

DigiPen Senior (CSBSRTIS)

http://seanmiddleditch.com/

2D Experience @ DigiPen: Subsonic, Core

Today's Talk

2D game rendering tips and techniques

Generally applicable to both D3D and OpenGL

- Any code mentioned will be for both

Optimization, effects

2D Graphics

66% as many D's as 3D

Why Go 2D?

2D is *much* simpler to code than 3D

2D art is *much* simpler to make

2D gameplay is *much* simpler to design/iterate

2D offers unique styles 3D does not

Raster Graphics vs Vector Graphics

Raster Graphics are techniques that draw existing rasterized images onto the screen

Almost all 2D games use raster graphics

Vector Graphics are techniques that use definitions of shapes to procedurally generate the final rasterized image on screen

Space Wars, Geometry Wars, Asteroids, etc.

Vector Graphics

Vector graphics are excellent for teams without artists

Very popular on older hardware due to limitations of the graphics capabilities

Vector graphics can be procedurally generated, morphed, and decomposed

Vector Graphics

Vector graphics styles are sometimes achieved with raster graphics techniques (like Subsonic)

Vector graphics are resizable procedurally with no loss of quality

Vector formats exist like SVG, tools exist like Illustrator or Inkscape

Raster Graphics

Also often just called **Bitmap Graphics**, which honestly is a less confusing term (vector graphics are *rasterized* during rendering)

Raster graphics are when *pre-rasterized* images are used

Rasterization is a process that creates a 2D pixel image

Raster Graphics

Raster graphics are vastly more popular than vector graphics

Raster graphics allow detailed images to be used for scenes, allowing potentially photo-realistic scenes

Vector graphic stylizations can be used in raster graphics

Bitmaps, Textures, Images, & Sprites

Bitmaps are 2D pixel **images** (not to be confused with Microsoft's Bitmap file format, which is just one of many bitmap format)

Textures are a common name for bitmaps in 3D APIs

Sprites are bitmap image textures that can be moved on screen or manipulated by a program

Sprites

Sprites are 2D objects you can move around

A static background image is not a sprite, nor are any objects drawn onto that image

2D games are often just called "sprite games" and the code often refers to all graphics images as "sprites" even if they are static images (yay for terminology consistency)

More on Sprites

A sprite is used for any object in game

The player is a sprite, each enemy is a sprite, every item is a sprite, a manipulatable door is a sprite, etc.

In other words, every game object that can be drawn is a sprite, or has a sprite component

Tiles

2D **tiles** are a common way of rendering levels

Each tile is an individual image drawn in a grid

Sometimes tiled levels are just a grid of sprites, sometimes they are drawn by specialized routines

Tiles are flexible, easy, and well understood

Batching

Batching is a technique allowing many sprites to be drawn with a single draw call

It requires **texture atlasing** in order to work

More details later in the talk

Layering

2D games often utilize some form of **layering** to determine draw order

For example, tiles may always be drawn behind the player

Easily handled using **Painter's Algorithm** in 2D, no need for z-buffering like 3D

Particles

Particles are used for a great many VFX (visual effects) in games, both 2D and 3D

Fire, smoke, dust, lasers, water droplets, blood, crumbling objects, and so on all use particles

Particle systems can range from simple to highly advanced, depending on specific needs

Shaders

Shaders are programs that run on the GPU, generally written in a high-level C-like language (HLSL, GLSL)

OpenGL 2.1 does not require shaders (it supports **fixed function** mode), but both Direct3D11 and GL|ES require user-written shaders

Shaders don't need to be hard or complex

Programming Overview for 2D

Writin teh codez for teh 2D

DirectX vs OpenGL

I heartily recommend D3D11 over OpenGL for students first learning graphics programming

- *Much* better debugging support
- *Much* better developer tools
- Better teaches vital hardware concepts
- Cleaner API with 98% fewer 1980's-isms

OpenGL of course is required if you care about porting to OSX, Linux, iOS, Android, NaCl, etc.

DirectX vs OpenGL Continued

If you're already familiar with one, try the other

- Good graphics programmers know both

If you're set on OpenGL, consider GL|ES

- Most of the OpenGL games industry is using GL|ES, not regular OpenGL, as iOS/Android/NaCl all use GL|ES exclusively

- Try using ANGLE (GL|ES on top of D3D9)

2D Cameras

Cameras define where the viewer is at and how the scene is **projected** onto the viewport (window)

2D games generally use an **orthographic** projection, which means there is no perspective correction (objects do not get smaller as they get further away)

Orthographic Cameras

In Direct3D using DirectXMath:

XMMatrixOrthographicLH

In OpenGL using GLU and the matrix stack:

glOrtho

In OpenGL using GLM and uniforms:

glm::gtc::matrix_projection::ortho

A Note on Handedness

DirectX typically uses a **left-handed** coordinate system

OpenGL typically uses a **right-handed** coordinate system

Matrices, Handedness, Pre and Post Multiplication, Row vs Column Major, and Notations (from my website)

A Note on Origins

Direct3D11 puts (0,0) at the *upper* left corner of screen space and texture coordinates

OpenGL puts (0,0) at the *bottom* left corner of screen space and texture coordinates

Makes it a pain to write code (especially 2D) that transparently supports both APIs

Textures

Textures are bitmap images loaded onto the GPU

The graphics APIs don't load images off disk for you, so another library is necessary

- Direct3D includes D3DX that does this
- OpenGL has no extension for loading data, but SDL, SFML, and GLFW all support it

Texture Creation

If you are manually creating textures (e.g. for an atlas)

With Direct3D11:

ID3D11Device::CreateTexture2D

With OpenGL:

glGenTextures, glTexImage2D, glBindTexture, glTexParameter

Drawing

Graphics hardware works with the concept of vertex buffers

Memory allocated on the GPU that contains the list of vertices to render

Must generate a list of vertices to draw, upload them to the GPU, and then tell the GPU to draw the vertices

Primitives

A **primitive** is the type of shape being rendered from vertices

A **triangle** requires 3 vertices, a **line** requires 2, etc.

Some APIs support **quads**, which seem ideal for 2D, but are sometimes not natively supported by hardware - always use 2 triangles

Vertex Buffers

Vertex buffers are allocated chunks of memory that have a specific **layout**

The layout specifies what data is in the buffer

- ex: position (float x2), UV (float x2), color (byte x4)

Previous example is an **interleaved** buffer, which is what you should always use

Creating Vertex Buffers

Vertex buffers have a size, some flags, and possibly initial data (e.g. for static buffers)

Direct3D11:
    ID3D11Device::CreateBuffer

OpenGL:
    glGenBuffers, glBufferData

Specifying Layout

OpenGL has a lot of different ways to do it, but OpenGL 2.1 specifically is simple (if confusing)

In Direct3D11:
ID3D11Device::CreateInputLayout

In OpenGL:
glVertexAttribPointer

Layout in Direct3D

Direct3D uses the concept of **semantics** to map vertex data (like the position) to the pipeline

Layout will specify "POSITION" to set the position, for example

Shader code uses the semantic names to map vertex attributes

Layout in OpenGL

OpenGL uses **slots** to bind vertex attributes

Slots are not specified in shaders explicitly in OpenGL 2.1 or GL|ES (but newer versions do thankfully support this)

Do not use glGetAttribLocation

Use glBindAttribLocation

Layout in OpenGL Continued

glVertexAttribPointer does not actually specify a pointer

The last pointer argument is the offset into the vertex buffer the attribute is at (e.g., UV coordinates comes after float2 position, so its offset is `2*sizeof(float)`, cast to `void*` as `reinterpret_cast<void*>(2*sizeof(float))`

Draw Commands

The draw command is a function that tells the GPU to render the vertices in a buffer as a particular type of primitive

In Direct3D:

ID3D11DeviceContext::Draw

In OpenGL:

glDrawArrays

Batching & Atlasing

Making rendering fast

Draw Call Overhead

Every draw call has a lot of overhead

Has to do with the API, driver, and especially the hardware

Changing state (current textures, shaders, etc.) is also expensive, and naturally implies more draw calls

Hardware Draw Call Overhead

GPU hardware has some number of shader cores

GPU hardware cannot (for the most part today) run multiple programs at the same time

Rendering one sprite per draw call means a lot of wasted hardware potential, not to mention state change overhead

Batching

As many objects as possible should be drawn together with a single draw call (one **batch**)

This requires using the same state (bound textures and shaders)

Also requires putting multiple objects into a single vertex buffer, or using instancing

Instancing

Instancing is a way to draw the same object many times, with different properties (like position)

It *can* be a huge performance boost most cases

Somewhat more complex than non-instanced rendering, not natively support in OpenGL 2.1, not supported at all in GL|ES 2

Instancing Calls

Instancing requires shaders and special buffers

Basic call in Direct3D:

ID3D11DeviceContext::DrawInstanced

Basic call in OpenGL:

glDrawArraysInstanced

Batching without Instancing

Sort all sprites by **material** (set of textures and shader used), layer, and so on

Push all sprites using the same material into a VBO

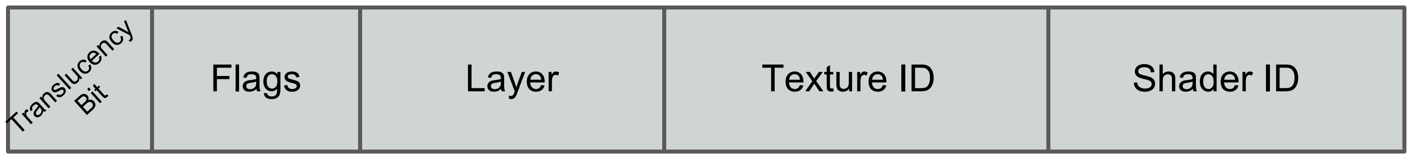
Set draw state

Draw the VBO

Sorting

Give each material (or individual texture and shader) a unique small-ish integer identifier

Compose a single 32-bit or 64-bit integer "key" for each object



Sorting Continued

Use radix sort

Integer key, pointer to sprite

Iterate over sorted list, render objects with the same texture and shaders (and other attributes that are part of state)

Tips for Sorting by State

If a sprite has only **transparency** (100% see-through) and **opaqueness** (0% see-through), render top layer to bottom layer

If a sprite has any **translucency** (anything other than 0% or 100% see-through) you must render back-to-front

Translucency bit in sort key and invert layer, sort automatically with no extra effort

Drawing Batches

```
For each sprite in sorted sprite list
    If sprite has a different state, render existing buffer and clear it
    Push sprite's triangles into buffer
Render buffer, if non-empty
```

Texture Atlases

Swapping textures requires changing state, and making more draw calls

Better to use as few textures as possible

Texture atlasing is a technique that combines multiple images into a single texture

Texture UVs

UV coordinates are the locations of the texture sampled at each triangle corner, interpolated over the triangle when rendered

Ranges from (0,0) to (1,1) in the common case

Sampling (0,0) to (0.1,0.1) would render one one-hundredth of the texture

Using Texture Atlases

Each image stored on a texture has a position in UV coordinates, e.g. (0,0) to (0.25,0.25)

Keep a data structure that knows locations of images in a texture

Render using these coordinates

Effectively allows more than one image per texture

Building Texture Atlases

The popular algorithm is an algorithm for bin-packing, originally created for lightmaps

Remember to put some padding between items

Ideally, do this at build time, but also quite feasible to do at runtime

Sharing Texture Atlases

Each sprite using any image in an atlas needs only that one texture

Possibly all your enemies (or even every sprite in the game) might fit in one atlas

Sort objects by texture/atlas to reduce state changes and reduce draw calls

Particles

Fancy effects for even simple games

Particles

Particles are little objects (usually sprites) that are generated by a **particle emitter** and all similar particles are part of a **particle system**

Particles are used for *tons* of effects in games, including: smoke, fire, water, dust, crumbling blocks, magical effects, lighting effects, and countless other things

Particle Emitters

A particle emitter is an object that creates particles with various properties

Properties that can be randomized include size, velocity, acceleration, speed, color, lifetime, and many more

Other properties might be controlled by a curve or algorithm, such as particles that fade out as they age, or change from one color to another

Particle Systems

The particle system's job is to manage all the particles that have been created that have the same set of properties and algorithms

Particle systems are often just part of emitters rather than shared

Particle system updates particles lifetime, interpolates any properties that change with time, and renders particles

Particle Rendering

Particle systems should be rendered as a whole

Ideally, all particles in a system share any similar state that requires separate draw calls (so all particles in a system can be drawn with one draw call)

Very common to have translucency; use additive blending so sorting is not required between individual particles

Particle Vertex Buffers

Do not try to cull or intelligently handle individual particles

Stuff all particles from an emitter (or similar emitters) into a VBO and draw it

Do not use multiple VBOs for different properties; seems easy, slow rendering

Updating Particles, Simple

Easiest, fast method (but least flexible) is to write one C/C++ function for each type of particle you have

Run that on each particle in the system, using the particle's current state and the current time as input

Requires only one iteration per particle system per update

Updating Particles, Flexible

A more advanced system creates a controller for each particle property per system

Controller can have curve and interpolation data

This is more work, and it's not easy to optimize

Lets you quickly create crazy new particle systems on the fly with little effort, valuable if you plan on lots of fancy effects in your game

Updating Particles, Fastest

Runtime generate CPU or GPU code from controllers

Not really feasible for student games in general

Don't spend time on trying this, just be aware of it (might be a good solo project for GAM350+ students?)

Large Particle Systems

Large particle systems can require a lot of time updating the vertex buffer

Can update sections of the system and draw them; more draw calls, but less latency

Update sub-resource ranges of the buffer, e.g. draw particles 0-99,999 while updating 100,000-199,999 or use "double buffering" for the particle vertex buffers

Field Particles

An alternative to using curves and explicit algorithms is to use fields of attractors and repellers

Demo: JavaScript Particle System

Very interesting way to visually build particles, even though curve-based systems are more popular in the industry

Fluids

Particles are one way to handle fluids in 2D

Demo: Fluid Particles

There are other (possibly better) fluid systems, but this is simple to do

More Demos

Smoke

Simple Particle Systems (Java)

Asteroids Benchmark

Stardust (from the magnanimous Allen Chou)

Keep it Simple

Very simple particle systems can do a lot

Start simple, build up if and only if you need to

Yes, I know I'm starting to sound like a broken record on this point, but it's really important

KISS - Keep It Simple, Super-dev

Shaders

Programming the GPU

Shader Basics

Shaders are programs that run on the GPU

- **Vertex shaders** transform vertices
- **Fragment shaders** render pixels on the screen
- **Geometry shaders** generate new primitives
- **Tessellation shaders** are specialized hardware for enhancing content
- **Compute shaders** are for GPGPU uses

Important Shaders

Odds are you will only use or need vertex shaders and fragment shaders (a.k.a. pixel shaders in Direct3D parlance)

Vertex shaders often just multiply the camera and projection matrices with each triangle's vertex and pass through the other attributes to the fragment shader, specially in 2D games

Fragment shaders often do all the cool stuff

A Requiem for Fixed Function

Old graphics hardware was **fixed function** meaning it was not programmable

D3D9 and OpenGL (before Core profile) have fixed function support and shaders are optional, but D3D10+, OpenGL Core, and GL|ES all require shaders

Fixed function made simple stuff slightly easier when getting started; otherwise, it's a dead end

Shading Languages

Two major families of shading language, with four primary dialects, and numerous versions

HLSL and **Cg** are very similar, from Microsoft and NVIDIA

GLSL and **ESSL** are for OpenGL and GL|ES

All very similar, C-like languages

Picking a Language

Not many choices

Direct3D only uses HLSL

OpenGL and GL|ES only use GLSL and ESSL respectively

Cg cross-compiles to HLSL or GLSL, but not ESSL; good on desktop, not on mobile

Sample GLSL Shaders

```
// VERTEX SHADER
uniform mat4 u_CameraProjection;
attribute vec2 a_Position;
attribute vec4 a_Color;
varying vec4 v_Color;

void main() {
  gl_Position = u_CameraProjection * vec4(a_Position, 0,
1);
  v_Color = a_Color;
}

// FRAGMENT SHADER
varying vec4 v_Color;

void main() {
  gl_Color = v_Color
}
```

"Advanced" Shaders

HLSL Shaders & Tutorials

GLSL Shaders & Tutorials

Many of these are geared towards 3D, but should give you a jumping point for learning how to use more shader features and tricks

Pixel Perfection

Old-school look and feel

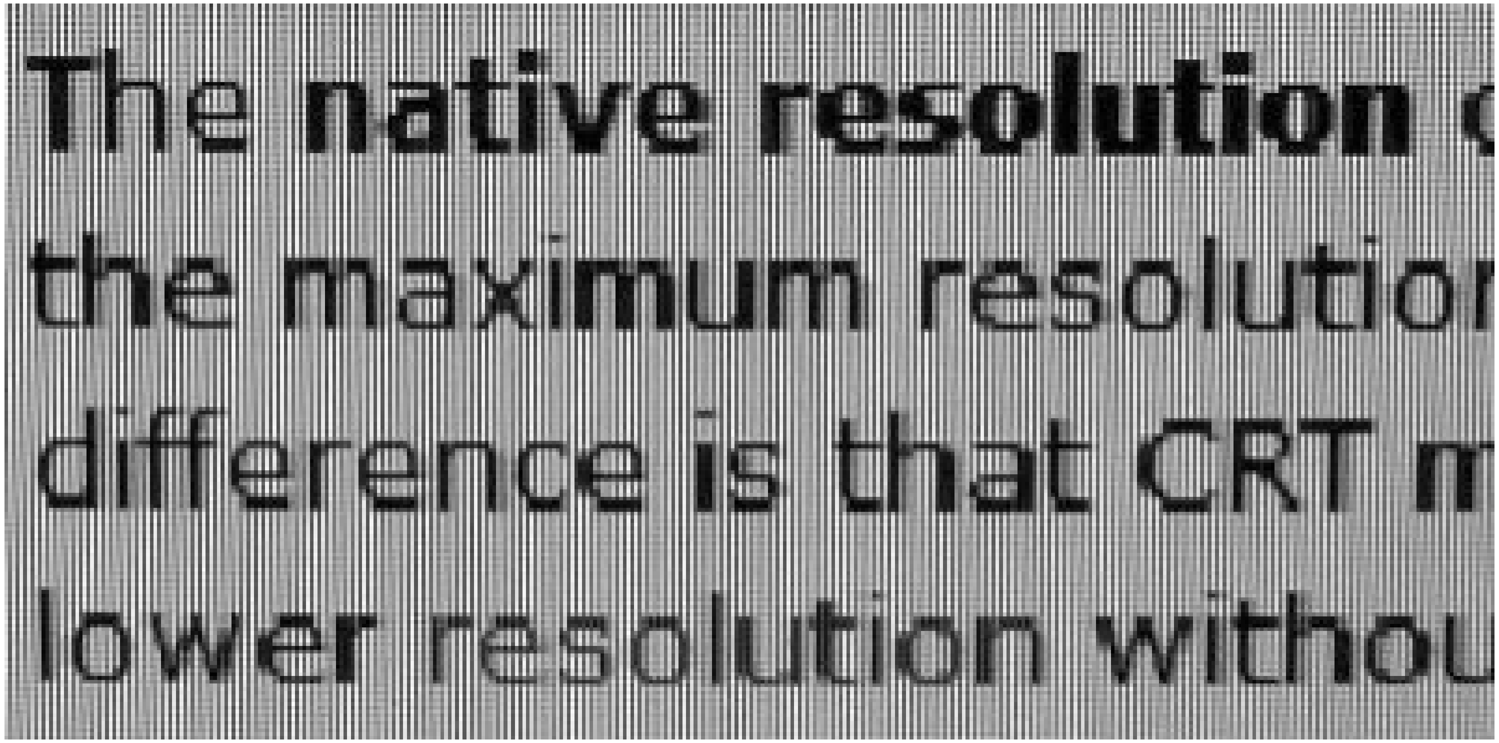
Pixel Perfect

Pixel perfect means that each pixel an artist painted in an image is mapped to exactly one pixel on the screen

Antialiasing and bad texture sampling can make images on the screen look blurry and ugly

Bad Pixels Example

Games can look this bad, too


The native resolution
the maximum resolution
difference is that CRT m
lower resolution withou

Pixel Centers

Each pixel on the screen is a square, not a point

(4,4) is not the center of the pixel 5 units from the upper left (or bottom left, in GL) corner

- (4.5,4.5) is the center of that pixel

Except in Direct3D 9 and earlier, which is dumb

More Pixel Centers

You mostly don't need to worry about any of this in D3D11 or OpenGL

If you're using D3D9 (it's 2012, please don't) then you'll need to account for the half-pixel offset

Sometimes the same issue pops up in other higher-level APIs, so be aware of it just in case

Scaling

Scaling can introduce artifacts

A 64x64 image scaled 20% will be 76.8x76.8

Naturally, that doesn't directly fit the pixel grid, and will look bad

If possible, scale by whole numbers, or fractional powers of two if the texture is a power of two

Rotating

Rotation also adds pixel artifacts, unless it's a perfect multiple of 90˚

Don't rely on angle and sin/cos math if you want pixel perfection, errors can accumulate

Use fixed rotation matrices, e.g.

```
rot(˚)     = [ 1,  0;  0,  1]
rot(90˚)   = [ 0, -1;  1,  0]
rot(180˚)  = [-1,  0;  0, -1]
rot(270˚)  = [ 0,  1; -1,  0]
```

Position

If an object is drawn at position (5.153,2.837) then it will not be drawn pixel perfect

Snap objects to whole pixels (or half-pixels, for D3D9) to ensure pixel perfection

Remember that all these can be done in your vertex shader, not just in the CPU

Summary

Finishing it up

Take Away

Use **batching** and **texture atlases** for drawing

Minimize draw calls and state changes

Use **particles** and **shaders** for advanced effects

Keep it simple

Questions & Answers

Thank you!