Game Architecture

User Interface

Today’s Agenda

- UI: what it is and why you’ll spend way more time on it than seems right
- Architecture
- Layout
- Rendering techniques
- Text
- Some third party options
- Homework assignment

What are we talking about?

- Frontend
- HUD
- Pause menus
- The ‘macro game’
- Integrated developer tools

Why it’s hard

- If it does it job right, probably don’t notice it
- UI does not have a lot of eyes on it
  - Special shortcuts used to bypass UI during development
- UI is involved in error handling
  - Rarely well tested
  - Certain behavior mandated by platform owners
- UI is very multi-disciplinary
  - Lots of room for misunderstanding
  - Often slow iteration times
- UI is a magnet for over-engineering
  - Getting data in and out of UI is often over-thought
- UI does not get respect
  - *Game Engine Architecture* does not cover it
  - Often where new programmers start their career

UI Architecture

- Widgets
- Events
- Animation
- Retained mode
- Immediate mode
- Model-View-Controller
- Model-View-View-Model
- Thoughts

UI Architecture

- **Widgets**
- Events
- Animation
- Retained mode
- Immediate mode
- Model-View-Controller
- Model-View-View-Model
- Thoughts

UI is composed of *widgets* or *controls*.

- Label
- Button
- Scroll bar
- List
- Etc.

Often parent-child relationship. Properties are often inherited from parent.

- Transform
- Color

UI Architecture

- Widgets
- **Events**
- Animation
- Retained mode
- Immediate mode
- Model-View-Controller
- Model-View-View-Model
- Thoughts

Events are the stuff that happens to UI:

- User taps screen, or presses button
- Player takes damage
- Screen resizes

Events may flow down a tree of widgets. Parents may have option to consume event or pass along to children.

UI Architecture

- Widgets
- Events
- **Animation**
- Retained mode
- Immediate mode
- Model-View-Controller
- Model-View-View-Model
- Thoughts

Game UI loves beautiful transitions and loops. Animation is key.

- Transform
- Color and opacity
- Particle systems

Components of a basic UI animation system.

- Animatable properties
- Keyframes
- Linear and Bezier interpolation
- Ability to sync business logic to animation

UI Architecture

- Widgets
- Events
- Animation
- **Retained mode**
- Immediate mode
- Model-View-Controller
- Model-View-View-Model
- Thoughts

- UI holds long lived state.
- Widgets have lifecycle.
  - Initialize
  - Update & handle events
  - Destroy
- Widgets = special case entities.
- How can widget implementation be decoupled from gameplay logic & data?

UI Architecture

- Widgets
- Events
- Animation
- Retained mode
- **Immediate mode**
- Model-View-Controller
- Model-View-View-Model
- Thoughts

API is like OpenGL. Every frame you say what UI you need:

```
if (button(“click me”, 100, 100))
{ /* handle the click */ }
```

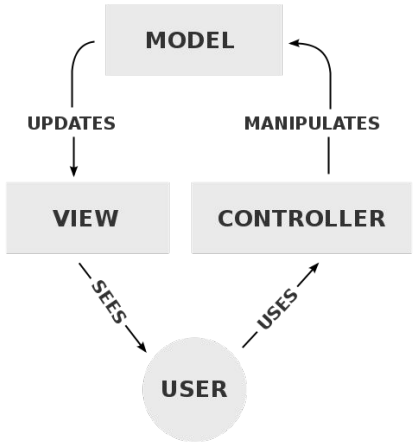
If you don’t say something, it won’t exist. No UI state lives longer than 1 frame.

UI Architecture

- Widgets
- Events
- Animation
- Retained mode
- Immediate mode
- **Model-View-Controller**
- Model-View-View-Model
- Thoughts

How do we decouple gameplay logic from widget implementation?

- Model - Gameplay logic/data
- View - The presentation
- Controller - Input processing

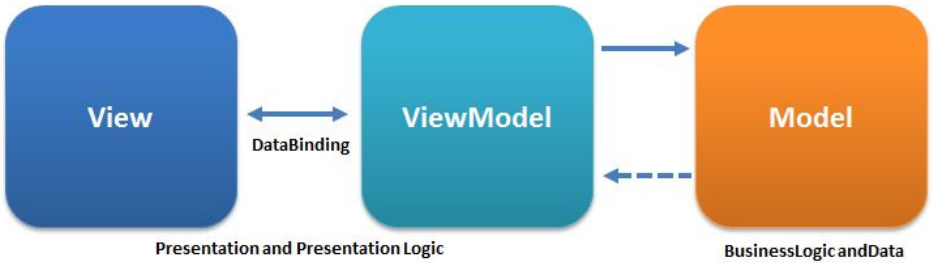


UI Architecture

- Widgets
- Events
- Animation
- Retained mode
- Immediate mode
- Model-View-Controller
- **Model-View-View-Model**
- Thoughts

How do we decouple gameplay logic from widget implementation?

- Model - Gameplay logic/data
- View - The presentation
- View-model - Binding layer



UI Architecture

- Widgets
- Events
- Animation
- Retained mode
- Immediate mode
- Model-View-Controller
- Model-View-View-Model
- **Thoughts**

- Use retained mode for complex UI
- Use immediate mode for simple UI
- Animation is essential
- Adapting MVC or MVVM formalism is not required -- just have a standard way of integrating UI with rest of your game

Layout

- Layout = Location of UI widgets on screen
- Layout must be pixel perfect
  - Single transform path for all widgets
  - Know your pixel center convention
    - Direct3D 9 = pixel center is 0,0
    - Direct3D 10+ = pixel center is 0.5,0.5
    - OpenGL = pixel center is 0.5,0.5
- Console games = fixed resolution = static layout
  - This is trivial.
- PC and mobile games = variety of resolutions = dynamic layout
  - Don’t want to manually specify layout for all possible resolutions.
  - This is the focus of the next couple slides.

Dynamic Layout

- UI is authored at some expected resolution (E_x, E_y).
- UI is displayed at some actual resolution (A_x, A_y).

Handy operations:

- X Scale: $Position_{Actual} = Position_{Expected} * A_x / E_x$
- Y Scale: $Position_{Actual} = Position_{Expected} * A_y / E_y$
- All Scale: $Position_{Actual} = Position_{ExpectedX} * A_x / E_x$, $Position_{ExpectedY} * A_y / E_y$
- X Anchor: $Position_X$ += $Position_{\underline{ParentX}}$
- Y Anchor: $Position_Y$ += $Position_{\underline{ParentY}}$

Layout Transform

World space widget transform = *[Pivot] * [Translation] * [Scale] * [Rot] * [Parent]*

- *[Pivot] = -Widget$_{Pivot}$ * Widget$_{Bounds}$*
- *[Translation] = Widget$_{Position}$ * Dynamic_Scale + Dynamic_Anchor*
  - Dynamic_Scale is scaling induced expected resolution != actual resolution
  - Dynamic_Anchor is locating a widget relative to expected resolution
- *[Scale] = Widget$_{Scale}$*
- *[Rot] = Widget$_{Rotation}$*
- *[Parent]* = Parent widget local transform

Rendering techniques

UI wants smooth, pixel perfect edges. Even if we scale.

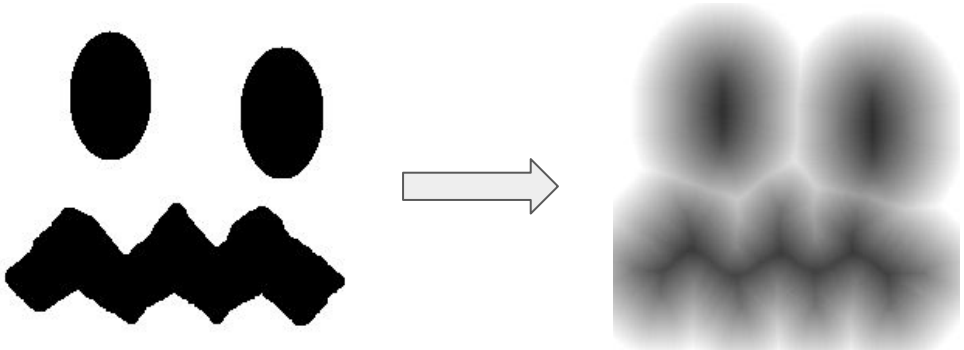
- Bitmap graphics - mapping texture onto geometry
  - E.g. how the rest of your game renders stuff
  - Alpha tested textures = widget edges
  - Really big textures so we never upscale
  - Or, signed distance fields
- Vector graphics - analytically describing edges
  - Parametric equations describe edges
  - Scaling (and other transforms) are always perfect
  - Rendering and authoring is more complicated

Signed Distance Fields

Richard Mitton’s description:

“Imagine you've got a black-and-white image, where let's say the black parts are considered *inside*, and the white parts are considered *outside*. What you want is a quick method of looking up how far it is from any given point to the inside.

“A SDF is just an image where each pixel contains the distance to the nearest point on the boundary. So if a pixel is *outside*, it'll contain maybe +10 if it's 10 pixels away. If it's *inside*, it'll contain -10.”



Signed Distance Fields Part 2

To draw widget with SDF:

- Enable linear interpolation for textures
- Enable alpha test
- In fragment shader:
  - If sampled SDF value <= 0, draw
  - Else, discard

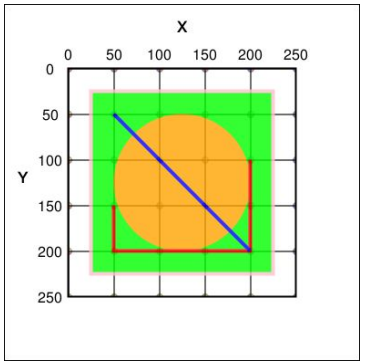
GPU can smoothly interpolate gradients like those in SDF, so edges are perfect.

Vector graphics

Use domain-specific, high level rendering primitives:

- TrueType font - Bezier control points
- Scalable Vector Graphics (SVG) - Basic shapes

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <rect x="25" y="25" width="200" height="200" fill="lime" stroke-width="4" stroke="pink" />
  <circle cx="125" cy="125" r="75" fill="orange" />
  <polyline points="50,150 50,200 200,200 200,100" stroke="red" stroke-width="4" fill="none" />
  <line x1="50" y1="50" x2="200" y2="200" stroke="blue" stroke-width="4" />
</svg>
```



Source: Wikipedia

Vector Graphics Part 2

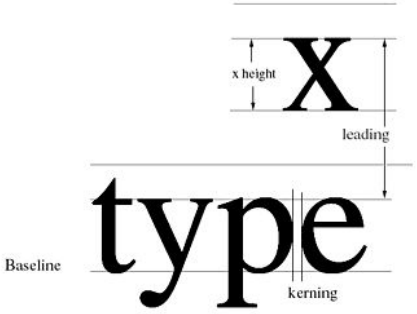
Usual architecture:

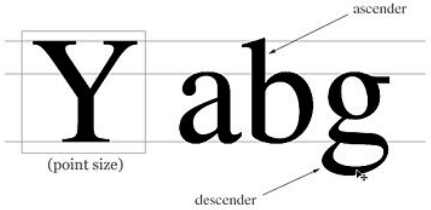
- Convert high level primitives into textures
  - Usually done on CPU
- Conversion is generally expensive
  - Maintain a texture cache
- Render with alpha test enabled

Text: Font

Know your terms:

- X-height
- Baseline
- Leading
- Kerning
- Ascender
- Descender
- Tracking
- Typeface





Serif

A serif is a small decorative flourish on the end of the strokes that make up letters and symbols

Sans Serif

"Sans" (to be without) Serif fonts do NOT have any flourishes at the end of strokes.

Text: Rendering

Two main options:

1. Bitmap graphics
    a. Offline, create atlas texture with all necessary letters and symbols.
    b. Use texture coords to select specific symbols.
    c. Consider signed distance fields to scale more cleanly.
2. Vector graphics.
    a. Render directly from TrueType with caching.
    b. Consider stb_truetype.

May want to add support for simple markup:

- Change text color / other properties.
- Render special symbols such as controller buttons.

Text: Localization

- Localization = preparing to ship outside your native culture, language, etc.
- Text considerations:
  - Supporting all the necessary symbols/letters
    - For bitmap fonts, do you need more than one texture?
  - Supporting the necessary display conventions (right-to-left, vertical)
  - Affordances for much longer/shorter words (*cough* German *cough*)
- Know Unicode:
  - Assigns a 32-bit integer to **every** written language symbol (code points)
  - Unicode has several standard encodings:
    - UTF-8 - Variable length, if high bit is set, look at next byte. **First 127 points -> ASCII.**
    - UTF-16 - 16-bit, fixed length, covers all the Unicode you will ever need.
    - UTF-32 - You do not need this.

Implementing Text Localization

1. Offline:
    a. Choose an encoding
        i. UTF-8 or UTF-16
    b. Find all the places in code and data where user-visible strings are stored
        i. Try to avoid strings baked into other game textures
    c. Build a big list of those strings
    d. Get them translated
    e. For bitmap fonts, build a library of all necessary symbols and pack into texture
    f. Find min,max extents of each string and use to size UI elements
2. At runtime:
    a. Detect current language
    b. Use indirection / patching to map native strings to translated strings

Third Party Options

Some useful packages:

- Scaleform - Flash player for games commonly used for UI
- Stb_truetype - Small, open source TrueType renderer
- IMGUI - Small, open source immediate mode UI

Homework

Create some simple immediate mode widgets...

Summary

- UI is challenging
- Dynamic layout: expected and actual resolution, scale, anchor
- Bitmap rendering
- Vector rendering
- Localization

End Lecture