Lecture 6x

Scene Graphs
Recall: Structure of a CUGL Application

Main → Application

Active
Scene
Models
Root Node

Dormant
Scene
Models
Root Node
Recall: The Application Class

**onStartup()**

- Handles the game assets
  - Attaches the asset loaders
  - Loads immediate assets
- Starts any global singletons
  - **Example**: AudioChannels
- Creates any player modes
  - But does not launch *yet*
  - Waits for assets to load
  - Like `GDXRoot` in 3152

**update()**

- Called each animation frame
- Manages gameplay
  - Converts input to actions
  - Processes NPC behavior
  - Resolves physics
  - Resolves other interactions
- Updates the scene graph
  - Transforms nodes
  - Enables/disables nodes
Recall: The Application Class

**onStartup()**
- Handles the game assets
- Attaches the asset loaders
- Loads immediate assets
- Starts any singletons or models
- Sets any player modes
- But does not launch *yet*
- Waits for assets to load
- Like `GDXRoot` in 3152

**update()**
- Called each animation frame
- Manages gameplay
- Converts input actions
- Resolves NPC behaviors
- Resolves other interactions
- Updates the scene graph
- Transforms nodes
- Enables/disables nodes

**onShutdown()**
- Cleans this up
- Does not draw! Handled separately
Drawing in CUGL

- Use `render()` method
- Called after `update()`
- Clears screen first
- Uses clear color field

- Can use any OpenGL
  - Included in `CUBase.h`
  - Best to use OpenGL ES (subset of OpenGL)

- Or use a `SpriteBatch`
  - *Mostly* like in 3152

```cpp
void render(const sh_ptr<SpriteBatch>& batch) {
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glDisableVertexAttribArray(0);
}

void render(const sh_ptr<SpriteBatch>& batch) {
    batch->begin();
    batch->draw(image1, Vec2(10, 10));
    batch->draw(image2, Vec2(50, 20));
    batch->end();
}
```
The Scene Graph

Or any subclass
The Scene Graph

- Scene
  - Node
    - Bounded box inside
    - Game Camera
    - Coords relative to parent box

Node Graphs
Each Node is a Coordinate System

Scene Root

Node

Node

Node

Node

Node
Each Node is a Coordinate System
Each Node is a Coordinate System

Scene Root

Node

Node

Node

Node

Origin

Origin

Origin

Scene Graphs
Motivation: Touch Interfaces

- Touch handler requires
  - Which object touched
  - Location inside object
- Scene graph is a *search tree*
  - Check if touch is in parent
  - … then check each child
  - Faster than linear search
- But limit this to a *search*
  - No input control in node
  - Use polling over callbacks
Settings Pass Down the Graph

Scene Root

Transforms on parent also transform children
Settings Pass Down the Graph

Scene Root

Node

Node

Node

Transparency on parent also applies to children
Settings Pass Down the Graph

Disabling the parent also disables children
Anchors and Content

- Nodes have **content size**
  - Width/height of contents
  - Measured in node space
  - But only a guideline: content can be outside

- Nodes have an **anchor**
  - Location in node space
  - *Percentage* of width/height
  - Does not affect the origin

- Both may affect **position**
Anchors and Content

- Nodes have **content size**
  - Width/height of contents
  - Measured in node space
  - But only a guideline: content can be outside

- Nodes have an **anchor**
  - Location in node space
  - *Percentage* of width/height
  - Does not affect the origin

- Both may affect **position**
Anchors and Content

- Nodes have **content size**
  - Width/height of contents
  - Measured in node space
  - But only a guideline: content can be outside
- Nodes have an **anchor**
  - Location in node space
  - *Percentage* of width/height
  - Does not affect the origin
- Both may affect **position**
Anchors and Content

- Nodes have **content size**
  - Width/height of contents
  - Measured in node space
  - But only a guideline: content can be outside

- Nodes have an **anchor**
  - Location in node space
  - **Percentage** of width/height
  - Does not affect the origin

- Both may affect **position**
Anchor and Position

Anchor: (0,0)
Position: (150,50)
Anchor and Position

Parent

Anchor: (0.5,0.5)
Position: (150,50)

Node

Child

Child

Origin
Layout Managers

- Not all devices have the same aspect ratio
- Sometimes, want placement to adjust to fit
Layout Managers

• Not all devices have the same aspect ratio

• Sometimes, want placement to adjust to fit
Layout Managers

AnchorLayout

FlowLayout

GridLayout
Layout Managers

Node  Parent  Node
Node  Node  Parent
Node  Node  Node
Parent  Node  Node
Node  Node  Node
Node  Node  Node

See Documentation for Details

GridLayout

Scene Graphs
How to Use a Layout Manager

1. Create a layout manager

2. Assign a relative position to each child
   - **Example**: middle left in an anchor layout
   - Layout manager maps strings to layout
   - Use the “name” string of the child node

3. Attach manager to the parent node

4. Call `doLayout()` on the parent
Safe Area: Modern Phones

**UI elements** should avoid notch, rounded corners

But animations should fill screen
Safe Area: Modern Phones

Art that must fill the screen

See Display class to find safe area

Elements to stay in safe area
Rendering a Scene is Easy

- `scene->render(batch)`
  - Uses SpriteBatch to draw
  - Calls `begin()/end()` for you
  - Sets the SpriteBatch camera
  - Limits *in-between* drawing

- Uses a *preorder traversal*
  - Draws a parent node first
  - Draws children in order
  - Parent acts as background

Scene Graphs
Is Preorder Traversal Always Good?

Good for UI Elements

Bad For Animation

Scene Graphs
Is Preorder Traversal Always Good?

Good for UI Elements

Bad For Animation

More on this next lecture
Specialized Nodes

- CUGL has many node types
  - SpriteNode (animation)
  - WireNode (wireframes)
  - PolygonNode (tiled shapes)
  - PathNode (lines with width)
  - NinePatch (UI elements)
  - Label (text)

- Learn them outside of class
  - Read the documentation
  - Play with the demos
The JSON Language for Scene Graphs

```
"textfield" : {
    "type" : "Node",
    "format" : {
        "type" : "Anchored"
    },
    "children" : {
        "action" : {
            "type" : "TextField",
            "data" : {
                "font" : "felt",
                "text" : "Edit me",
                "size" : [600,80],
                "anchor" : [0.5,0.5]
            }
        },
        "layout" : {
            "x_anchor" : "center",
            "y_anchor" : "top"
        }
    }
}
```
JSON Language for Scene Graphs

```
"textfield" : {
    "type" : "Node",
    "format" : {
        "type" : "Anchored"
    },
    "children" : {
        "action" : {
            "type" : "TextField",
            "data" : {
                "font" : "felt",
                "text" : "Edit me",
                "size" : [600,80],
                "anchor" : [0.5,0.5]
            }
        },
        "layout" : {
            "x_anchor" : "center",
            "y_anchor" : "top"
        }
    }
}
```
JSON Language for Scene Graphs

Each node has

- Type
- Format
- Data
- Children
- Layout

Scene Graphs
# Using JSON Scene Graphs

## Advantages

- **Designers do not need C++**
  - Using special tool in lab
  - Tool good for entire semester
- **Format is ideal for mobile**
  - Integrated layout managers
  - Aspect ratio support is easy
- **Integration is simple**
  - Load JSON with asset loader
  - Refer to scene root by name

## Disadvantages

- **UI still needs custom code**
  - Buttons etc. do nothing
  - Essentially need listeners
  - Programmers do manually
- **Files can be very confusing**
  - Format is a tree structure
  - Each tree node is verbose
- **Not a level editor format!**
  - Levels need more info
Widgets: JSON Templates

**Widget**

```json
"variables": {
  "image": ["children", "up", "data", "texture"]
},
"contents": {
  "type": "Button",
  "data": {
    "upnode": "up", "visible": false,
    "anchor": [0.5, 0.5], "scale": 0.8
  },
  "children": {
    "up": {
      "type": "Image",
      "data": { "texture": "play" }
    }
  }
}
```

**JSON**

```json
"widgets": {
  "mybutton": "widgets/mybutton.json",
},
"scene2s": {
  "thescene": {
    "type": "Node",
    "format": { "type": "Anchored" },
    "children": {
      "button": {
        "type": "Widget",
        "data": {
          "key": "mybutton",
          "variables": { "image": "altplay" }
        }
      }
    },
    "layout": { "x_anchor": "center" }
  }
}
```

- Widget is a subtree
- Replace w/ subtree
Widgets: JSON Templates

### Widget

```json
"variables" : {
  "image" : ["children","up","data","texture"]
},

"contents" : {
  "type" : "Button",
  "data" : {
    "upnode" : "up", "visible" : false,
    "anchor" : [0.5,0.5], "scale" : 0.8
  }
},

"children" : {
  "up" : {
    "type" : "Image",
    "data" : { "texture" : "play" }
  }
}}
```

### JSON

```json
"widgets" : {
  "mybutton" : "widgets/mybutton.json",
},

"scene2s" : {
  "thescene" : {
    "type" : "Node",
    "format" : { "type" : "Anchored" },
    "children" : {
      "button" : {
        "type" : "Widget",
        "data" : {
          "key" : "mybutton",
          "variables" : { "image" : "altplay" }
        },
        "layout" : { "x_anchor" : "center" }
      }
    }
  }
}
```

- Full path to value to change
- Change the variable
- Provide the layout
One Last Problem: Physics
One Last Problem: **Physics**

How big is that scene graph?
## Defining Custom Nodes

<table>
<thead>
<tr>
<th><strong>draw()</strong></th>
<th><strong>generateRenderData()</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- Overridden to render node</td>
<td></td>
</tr>
<tr>
<td>- Only node, not children</td>
<td></td>
</tr>
<tr>
<td>- The <code>render</code> method (do not touch) handles children</td>
<td></td>
</tr>
<tr>
<td>- Drawing data is <strong>cached</strong></td>
<td></td>
</tr>
<tr>
<td>- The vertex positions</td>
<td></td>
</tr>
<tr>
<td>- The vertex colors</td>
<td></td>
</tr>
<tr>
<td>- The texture coordinates</td>
<td></td>
</tr>
<tr>
<td>- Cache passed to <code>SpriteBatch</code></td>
<td></td>
</tr>
<tr>
<td>- Overridden to update cache</td>
<td></td>
</tr>
<tr>
<td>- Change vertex positions</td>
<td></td>
</tr>
<tr>
<td>- Change vertex colors</td>
<td></td>
</tr>
<tr>
<td>- Change texture coordinates</td>
<td></td>
</tr>
<tr>
<td>- Only needed for <strong>reshaping</strong></td>
<td></td>
</tr>
<tr>
<td>- Transforms for movement</td>
<td></td>
</tr>
<tr>
<td>- Called infrequently</td>
<td></td>
</tr>
<tr>
<td>- Optimizes the render pass</td>
<td></td>
</tr>
</tbody>
</table>
The **draw()** Method

```cpp
void CustomNode::draw(const std::shared_ptr<SpriteBatch>& batch,
                      const Affine2& transform, Color4 tint) {

    if (!_rendered) {
        generateRenderData();
    }

    batch->setColor(tint);
    batch->setTexture(_texture);
    batch->setBlendEquation(_blendEquation);
    batch->setBlendFunc(_srcFactor, _dstFactor);

    batch->fill(_vertices, _vertsize, 0,
                 _indices, _indxsize, 0,
                 transform);
}
```
The **draw()** Method

```cpp
void CustomNode::draw(const std::shared_ptr<SpriteBatch>& batch,
                       const Affine2& transform, Color4 tint) {
    if (!_rendered) {
        generateRenderData();
    }

    batch->setColor(tint);
    batch->setTexture(_texture);
    batch->setBlendEquation(_blendEquation);
    batch->setBlendFunc(_srcFactor, _dstFactor);

    batch->fill(_vertices, _vertsize, 0,
                 _indices, _indxsize, 0,
                 transform);
}
```
Summary

- CUGL tries to leverage ideas from 3152
  - Top level class works like the classic GDXRoot
  - Design architecture to switch between modes
  - Use SpriteBatch class to draw textures in 2D.

- New idea is using scene graphs to draw
  - Tree of nodes with relative coordinate systems
  - Makes touch input easier to process
  - Also helps with animation (later)

- JSON language makes design easier