CPSC 427
Video Game Programming

Game Programming Basics: Event Driven Programming & Entity Component System (ECS)

Play
This year’s game theme

A) Non-violent games, for ‘kids’

B) Randomness and surprise!

C) Time counts, 10 seconds!
Register your Team!

Even if incomplete, please register

-> Canvas -> People -> Groups -> Team

• 11 people still without a team
• 6 teams with 5/6 members

-> need to form one more team and add 5-6 members
Entity Component System (ECS)

Summary and extensions

ECS is used in Minecraft and many other commercial games
Problem: associating entities and components

Mario
Goomba1
Luigi
Goomba2
Position
Velocity
Jumps
Player
Squishable

Object-oriented-programming (OOP)?

ECS = containers of components?
Memory & ECS

Where do we store our Components?

• Inside Entities?

Update loop has to access non-contiguous memory repeatedly!

Slow memory access!
ECS = std::map?

- Associate components to entities
- Dynamic!
- Fast?

<table>
<thead>
<tr>
<th>Task</th>
<th>(hash) map</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamically add/remove a component</td>
<td>insert, emplace, erase</td>
</tr>
<tr>
<td>Check if entity has N components</td>
<td>count</td>
</tr>
<tr>
<td>Get component of type X of entity</td>
<td>find</td>
</tr>
<tr>
<td>Iterate over all components of type X</td>
<td>begin() iter-&gt;second</td>
</tr>
<tr>
<td>Iterate over all entities with component X</td>
<td>begin() iter-&gt;first</td>
</tr>
</tbody>
</table>
Try std::map out for A0

We will release a template

```cpp
// A container that stores components of type 'Component'
// TODO: You will have to change this class to be applicable to different component types
class ComponentContainer {
private:
    // TODO: add variables to store components and to associate components to entities
public:
    ComponentContainer() {};

    // Inserts a component c associated to entity e
    // TODO: add insert functionality and define the right return type
    /*
    TODO insert(Entity e, Component c)
    
    assert(!has(e) && "Entity already contained in registry");
    */

    // Checks if entity e has a component of type 'Component'
    // TODO: add has functionality
    bool has(Entity e) {
        
    };

    // Removes the component of type 'Component' from entity e
    void remove(Entity e) {
        // TODO: add remove functionality
    };

    // Returns the component of type 'Component' associated with entity e
    // TODO: add get functionality, including the right return type
    /*
    TODO get(Entity e) {
    };
    */
```
Memory & ECS

Where do we store our Components?

• In a map?
  • *It has all the functionalities*

Update loop has to access non-contiguous memory repeatedly!

Slow memory access!

Memory Blocks

- position
- velocity
- collision
- sprite
Memory & ECS

Where do we store our Components?

- Array with holes?

Better cache utilization!

Not memory efficient!
Map + Dense Component Vectors (entity ID to component address index)

<table>
<thead>
<tr>
<th>ID</th>
<th>Position index</th>
<th>Jumps index</th>
<th>Squishable index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mario</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Goomba1</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Luigi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goomba2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Concept:** Combine dense vectors with a map

**Implementation:** `std::vector<Component>; std::map<Entity,unsigned int>`

Issues?
Map + Dense Vector (different visualization)
Each Component type has a *statically* allocated array

Minimizes costly cache misses

- *Keeps components we access around the same time close to each other*
Map + Component Vectors + Entity Vector Cache is Key

Update loop accesses contiguous memory **IDEAL!**
Map + Component Vector + Entity Vector

Registry for one component

Concept: Add a dense vector of entities to facilitate quick iteration over entities
Implementation: std::vector<Entities>; std::vector<Component>; std::map<Entity,unsigned int>

Easy to iterate over all velocity components that belong to an entity with a position

```cpp
for(int entity : velocity_entities) // using the entities array
    if (position_entity_map.has(entity)) // using the map
        position_entity_map.get(entity) += velocity_entity_map.get(entity); // using component array
```
Faster iteration via entity and component array

Accessing the velocity map (reg_velocity.map) is an unnecessary indirection

```java
for (int entity : velocity_entities) // efficient
    if (position_entity_map.has(entity)) // inefficient lookup
        position_entity_map.get(entity) += velocity_entity_map.get(entity); // 2x inefficient lookup
```

We can access the velocity components in linear fashion

```java
for (int vel_i = 0; vel_i < velocity_entities.size(); vel_i++) // efficient
    Entity entity = velocity_entities[vel_i]; // efficient
    int pos_i = position_entity_map.getIndex(entity); // inefficient lookup
    if (pos_i)
        position_components[pos_i] += reg_velocity_components[vel_i]; // efficient
```
ECS goals: fast & dynamic

- Associate components to entities
- Fast & dynamic

<table>
<thead>
<tr>
<th>Task</th>
<th>(hash) map</th>
<th>bitset</th>
<th>Comp. vec</th>
<th>Entity vec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamically add/rem. a comp.</td>
<td></td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check if entity has N components</td>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Get component X of entity</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Iterate over comp. of type X</td>
<td></td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iterate over ent. with comp. X</td>
<td></td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
If you want to take a deep dive…
Self-study: A special map approach
**Self-study: The ‘Sparse Set’**

**Concept:** Sparse array + dense array

**Implementation:**
- `std::vector<Entity> entities;`
- `std::vector<unsigned int> indices;`
- `std::vector<Components> components;`

---

<table>
<thead>
<tr>
<th>ID</th>
<th>Index Pos</th>
<th>Index Vel</th>
<th>Index Jump</th>
<th>Index Player</th>
<th>Index Squish</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Position | Velocity | Jumps | Player | Squishable
---|---|---|---|---

**Issues?**

---

© Alla Sheffer, Helge Rhodin
The map lookup ($map.get(entity)$) is costly
- A hashmap is $O(1)$, but that $1$ is big

Sparse set:
- An array as large as the number of entities in the game
  - Crazy waste of memory?!
  - 32 bit integer -> ???
  - A sparsely filled array
- A small dense array of all entities in sequence (as before)
  - Extremely fast lookup, insert, & clear
Assignments

Template framework

[File tree image with directories and files]
A1 – Game Graphics
A2 – Animation and Physics
Your project
Procedural Programming

**Sequential control flow**

- program performs a sequence of tasks & terminates
- good for physical simulation
- maintains consistent rendering frame rate

- difficult to model a long order of events
Event-Driven Programming

No main loop under your control

• vs. procedural

Control flow through event callbacks

• redraw the window now
• key was pressed -> react
• mouse moved -> react

Callback functions called from main loop when events occur

• mouse/keyboard
• ensures temporal order
• prevents concurrency
int main(int argc, char* argv[]) {
    if (!world.init(..)) {
        return EXIT_FAILURE;
    }
    while (!world.is_over()) {
        glfwPollEvents(); // process events
        world.update(); // update game state based on events + timer
        world.draw(); // render
    }
    world.destroy();
    return EXIT_SUCCESS;
}
Our game loop (A1-A3 Template, main.cpp)

```cpp
int main()
{
    // Global systems
    WorldSystem world;
    RenderSystem renderer;
    PhysicsSystem physics;
    AISystem ai;

    // Initializing window
    GLFWWindow* window = world.create_window();
    if (!window) {
        // Time to read the error message
        printf("Press any key to exit");
        getchar();
        return EXIT_FAILURE;
    }

    // initialize the main systems
    renderer.init(window);
    world.init(&renderer);

    // variable timestep loop
    auto t = clock::now();
    while (!world.is_over()) {
        // Processes system messages, if this wasn't present the window would become unresponsive
        glfwPollEvents();

        // Calculating elapsed times in milliseconds from the previous iteration
        auto now = clock::now();
        float elapsed_ms =
            (float)(std::chrono::duration_cast<std::chrono::microseconds>(now - t).count()) / 1000;

        t = now;

        world.step(elapsed_ms);
        ai.step(elapsed_ms);
        physics.step(elapsed_ms);
        world.handle_collisions();

        renderer.draw();

        // TODO A2: you can implement the debug freeze here but other places are possible too.
    }

    return EXIT_SUCCESS;
}
```
openGL

• Low-level graphics API
• C Interface accessed from C++
• Shaders – graphics
  • A LOT more details later
Even Callbacks

Set at start – in our template in world.init()

auto key_redirect = [](GLFWwindow* wnd, int _0, int _1, int _2, int _3) {
    ((World*)glfwGetWindowUserPointer(wnd))->on_key(wnd, _0, _1, _2, _3);
};

auto cursor_pos_redirect = [](GLFWwindow* wnd, double _0, double _1) {
    ((World*)glfwGetWindowUserPointer(wnd))->on_mouse_move(wnd, _0, _1);
};

glfwSetKeyCallback(m_window, key_redirect);
glfwSetCursorPosCallback(m_window, cursor_pos_redirect);

Another example would be a mouse click (same format)
Callback Actions

```cpp
void World::on_key(GLFWwindow*, int key, int, int action, int mod){
    if (action == GLFW_RELEASE && key == GLFW_KEY_R){
        ...
    }
}

void World::on_key(GLFWwindow*, int key, int, int action, int mod){
    if (action == GLFW_RELEASE && (mod & GLFW_MOD_SHIFT) && key == GLFW_KEY_COMMA){
        ...
    }
}

void World::on_mouse_move(GLFWwindow* window, double xpos, double ypos){
}
```