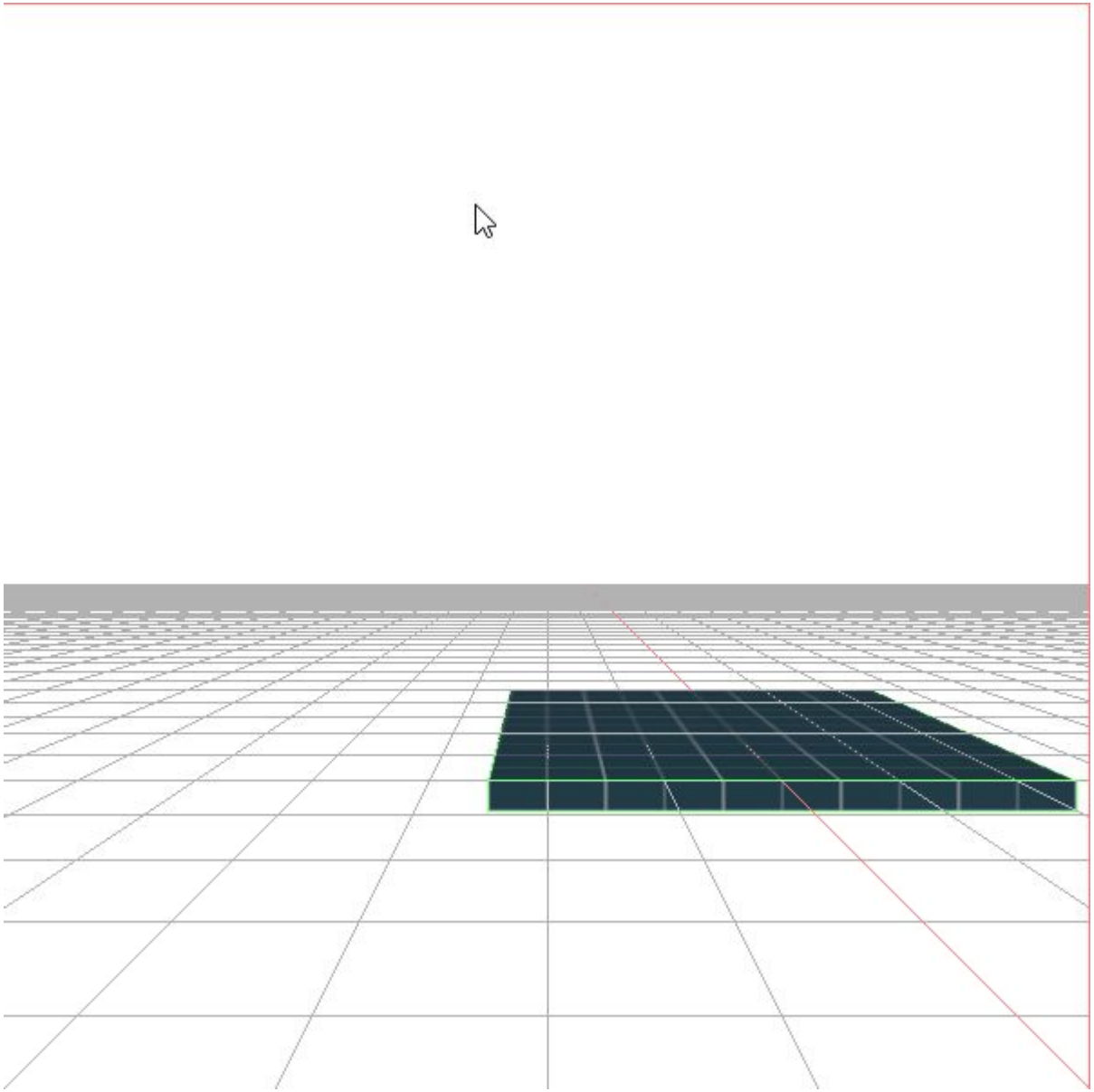
MVD: Engine Programming

07 - Scripts

alunthomas.evans@salle.url.edu

laSalle ENG
Universitat Ramon Llull



ScriptSystem

How would we go about making a Script System?

Embedding a scripting language

Embedding a scripting language is really neat.

It lets you write game code without having to re-compile the engine.

This allows Designers to programme game design without touching engine code-base.

It can be very easy (e.g. Lua has prebuilt C++ binaries)

It is quite a lot of work to make it 'worthwhile':

- you have to map every script function => C++ function

- you have write template functions to parse variables

You have to learn to code in another language!

Embedding Lua

```
//get Lua state from engine
lua_State *state = luaL_newstate();

//define a Lua function (2nd param)
//and link it with a c++ function (3rd param)
lua_register(state, "move_entity", move_entity);

// Load the script
luaL_loadfile(state, “move_entity_script.lua”);

// Execute the program by calling into it.
lua_pcall(state, 0, LUA_MULTRET, 0);
```

```
--Lua script
move_entity("floor", 0, 10, 0);
```

```
int move_entity(lua_State* state) {
    int args = lua_gettop(state);

    const char* ent_name = lua_tostring(state, 1);
    float x_pos = lua_tonumber(state, 2);
    float y_pos = lua_tonumber(state, 3);
    float z_pos = lua_tonumber(state, 4);

    int the_ent = ECS.getEntity(ent_name);
    Transform& the_trans = ECS.getComponentFromEntity<Transform>(the_ent);
    the_trans.position(x_pos, y_pos, z_pos);

    return 1;
}
```

Using templates to get variables

https://eliasdaler.wordpress.com/2013/10/11/lua_cpp_binder/

C++ “scripts”

We are going to create a Script System, and write our ‘scripts’ in C++

The disadvantage of this is that we have to compile every time we change our script

But the concepts behind creating a scripting engine are the same, and we save writing a lot of code.

If you want to implement Lua in your project, then go for it!

A script component?

We *could* make a script component, same as other components.

But we want would want to make custom components i.e. override base Script component

Our ECS doesn't work with pointers, and that means we can't store derived classes in it

```
typedef std::tuple<
std::vector<Transform>,
std::vector<Mesh>,
std::vector<Camera>,
std::vector<Light>,
std::vector<Control>,
std::vector<Collider>,
// this will only let us store Scripts,
// not classes derived from Script
std::vector<Script>
> ComponentArrays;
```

Go old fashioned

Create an array of ‘Script’ objects

Iterate all scripts, update each one in turn

```
class Script {
public:
    //constructor - must pass entity owner
    Script(int owner) { owner_ = owner;};

    //virtual function allows optional override
    virtual void init() {};

    // pure virtual functions FORCES execution of functions in derived classes
    virtual void update(float dt) = 0;

    //sets pointer to control system
    void setInput(ControlSystem* cont_sys) { input_ = cont_sys; };

protected:
    int owner_; //id of entity which owns this script
    ControlSystem* input_ = nullptr; //pointer to control system
};
```

- note virtual vs pure virtual function
- pointer to control system allows key/mouse access

Derived Script

```
class DerivedScript : public Script {
public:

    //MUST override constructor and call parent class constructor
    DerivedScript(int owner) : Script(owner) {}

    //MUST override update
    void update(float dt);

    //OPTIONALLY override init
    void init();

    //add whatever custom functions and properties here!
};
```

ScriptSystem

Stores an array of pointers to Script objects

```
std::vector<Script*> scripts_;
```

because we can add **derived (i.e. child)** Script classes to this vector

```
class ScriptSystem {
public:

    //initialize by setting control system pointer to send to scripts
    void init(ControlSystem* cont_sys) { input_ = cont_sys; };

    //lateInit calls init of all registered scripts
    void lateInit();

    //update all scripts
    void update(float dt);

    //register new script
    void registerScript(Script* new_script);

private:
    std::vector<Script*> scripts_;

    //pointer to the control system
    ControlSystem* input_;


};
```

```
//call init function of all registered scripts
void ScriptSystem::lateInit() {
    //init all scripts
    for (auto scr : scripts_)
        scr->init();
}

//update all scripts
void ScriptSystem::update(float dt) {
    for (auto scr : scripts_)
        scr->update(dt);
}

//register new script and pass script a pointer to control system
void ScriptSystem::registerScript(Script* new_script) {
    scripts_.push_back(new_script); //add to list
    new_script->setInput(input_); //tell script where control system is

}
```

Note registerScript sets the input component to allow script to access keys etc.

Disadvantage of our approach?

Disadvantage of our approach?

It's not data-driven! Pointers to scripts could be anywhere! Not taking advantage of processor cache

So it's slow! Use scripts for really custom level behaviour, or for prototyping

Otherwise it's better to hard-code the behaviour into a new component/system

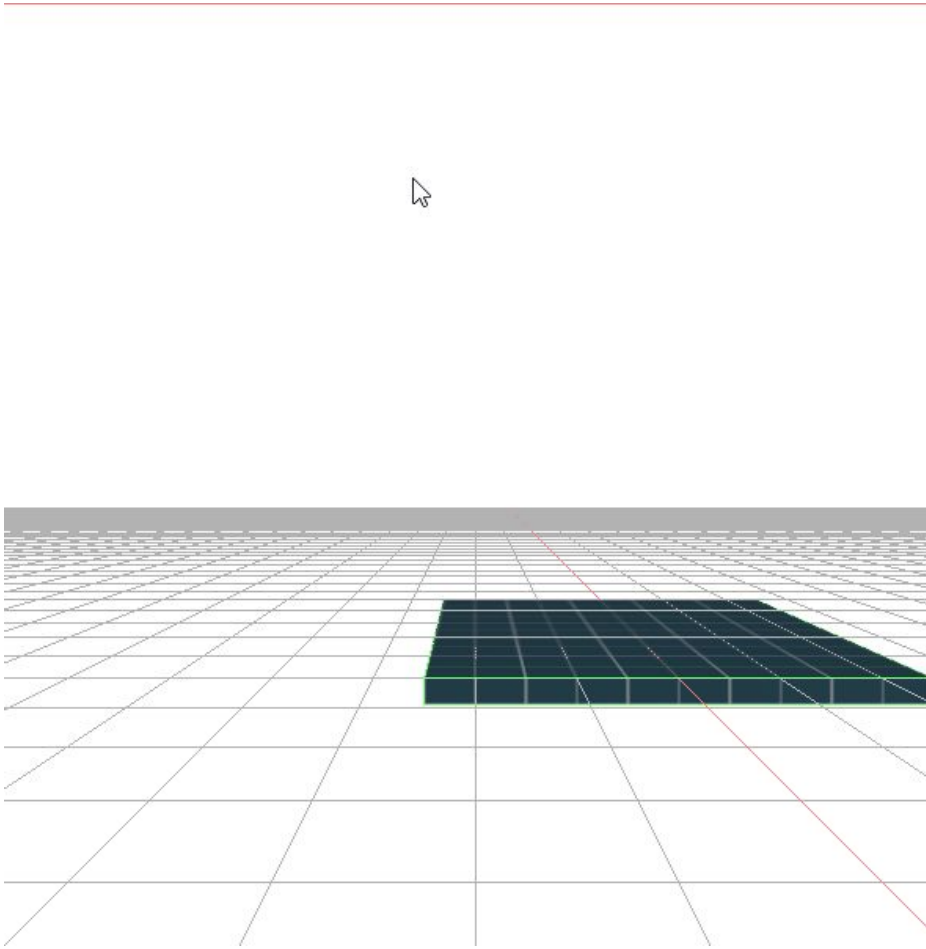
To make our demo

Task A:

- write a custom script that starts/stops moving a platform based on keys

Task B

- write a custom script that has a pointer to the move platform script, and calls functions in it to start/stop move



Advanced

Create a simple level with a series of platforms which move up and down automatically. The player must jump across them to reach a ‘final platform

Create a ‘target’ wall at end of level, which the user must activate by standing in front of it and pressing a key.

Target platform changes colour when activated

Christmas Project: Simple AI

Reading:

https://www.gamedev.net/articles/programming/artificial-intelligence/the-total-beginners-guide-to-game-ai-r4942/

Ideas:

- Simple pong
- Finite State Machine
  - patrolling guard
  - attack on sight
- Movement/pathfinding
  - A*
  - Navmesh