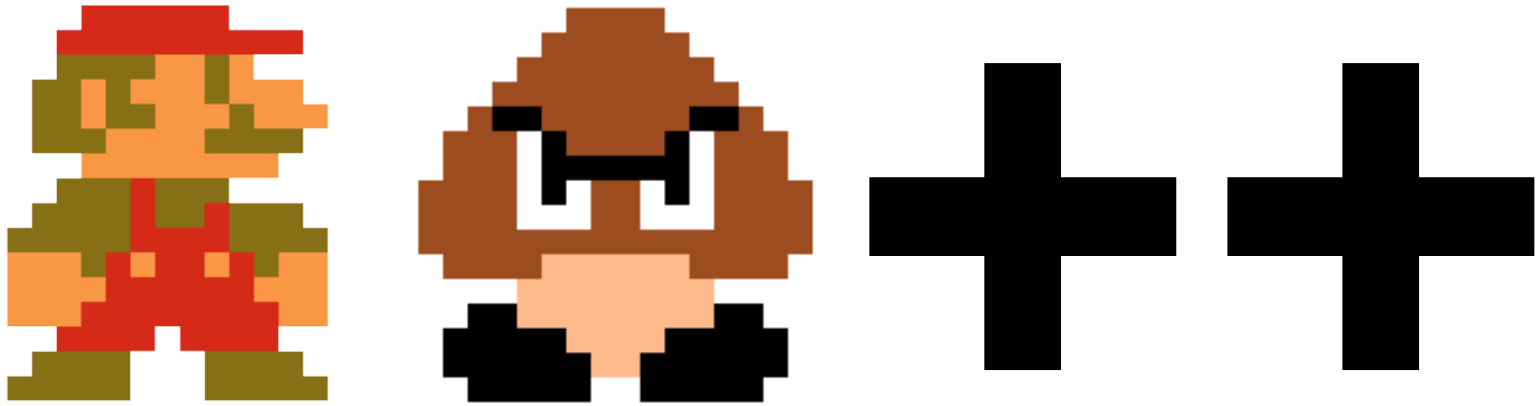
CPSC 427
Video Game Programming

ECS Views and BTrees in modern C++

Helge Rhodin

Advanced ECS



ECS – many small components

- ***Issues?***
  - *Many lookups*
  - *Cumbersome to write*

```
for (unsigned int i=0; i< ECS::registry<Position>.components.size(); i++)
{
    Position& position = ECS::registry<Position>.components[i];
    ECS::Entity entity = ECS::registry<Position>.entities[i];
    if (!ECS::registry<Velocity>.has(entity))
        continue;
    Velocity& velocity = ECS::registry<Velocity>.components[i];

    position += velocity;
}
```

Views with iterators (first try)

Iterators in C++: https://www.cplusplus.com/reference/iterator/

An iterator is any object that, pointing to some element in a range of elements (such as an array or a container), has the ability to iterate through the elements of that range using a set of operators (with at least the increment (++) and dereference (*) operators).

```
for (std::tuple<ECS::Entity, Position, Velocity> tuple : ECS::view<ECS::Entity, Position, Velocity>())
{
    ECS::Entity entity = std::get<0>(tuple);
    Position& position = std::get<1>(tuple);
    Velocity& velocity = std::get<2>(tuple);

    position += velocity;
}
```

Compact notation for iterating over all elements

- Do I really have to write Entity, Position, and Velocity 3x?

Views with iterators (second try)

```
for (auto tuple : ECS::view<ECS::Entity, Position, Velocity>())
{
    auto entity = std::get<0>(tuple);
    auto& position = std::get<1>(tuple);
    auto& velocity = std::get<2>(tuple);

    position += velocity;
}
```

- Relatively compact, but
- Requires good knowledge about iterators to implement view<…>()
- Annoying to write std::get<X>()
  - In C++ 17 you can use tuple unpacking (as in python)!

Views with callbacks

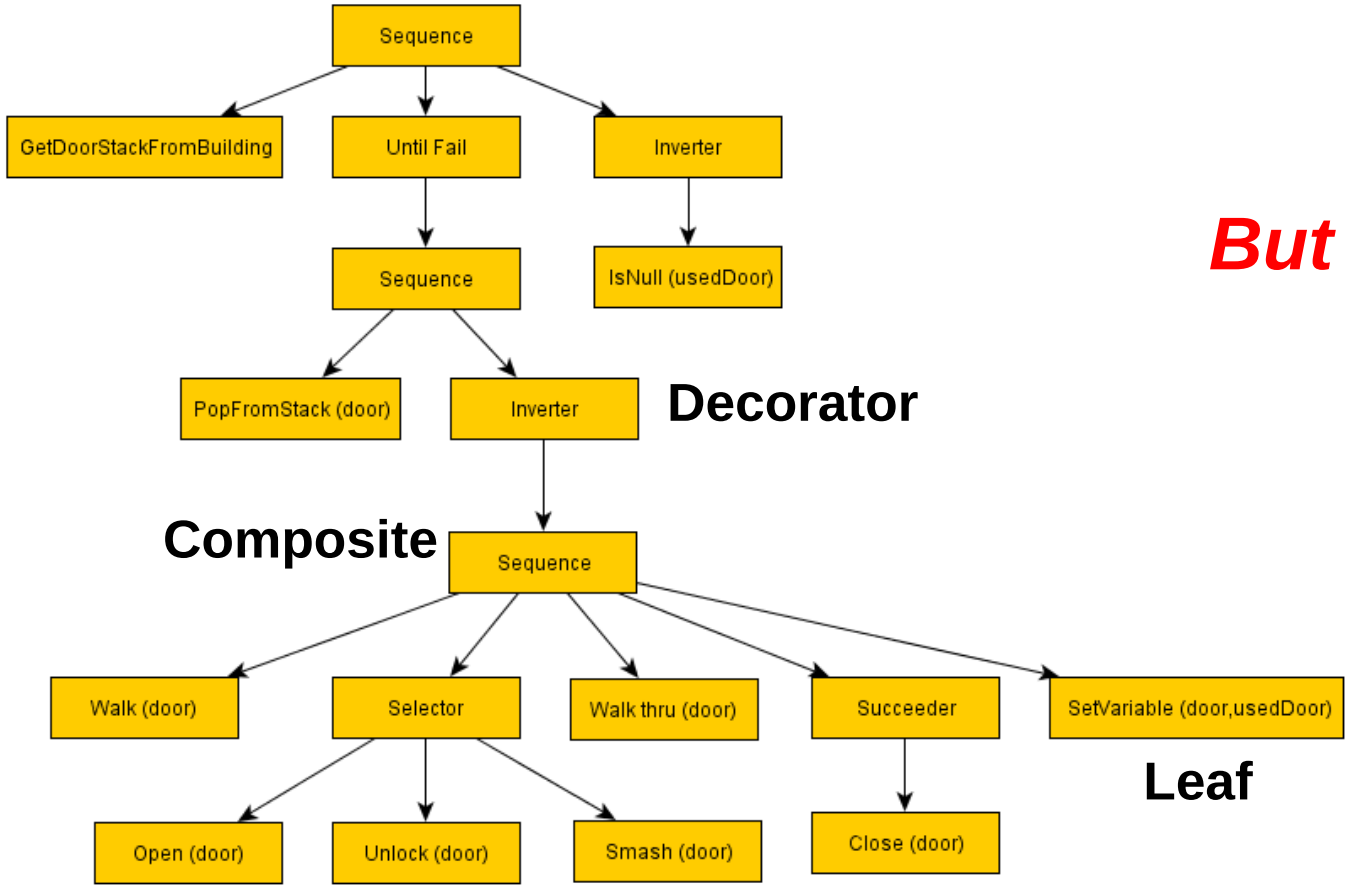
```
ECS::each<ECS::Entity, Position, Velocity>([](auto entity, auto& position, auto& velocity) {
    position += velocity;
    });
```

- Super compact!
- Relatively easy to implement each<…>(std::function<…> fun)
  - E.g. specialize implementation for two (or three) template arguments
    - Implement each with a for loop and has<Component>()
    - Takes a few lines of code, but only once in the tiny_ecs.h!
- Efficiency?
  - Internally, the map is accessed many times for has<Component>()
  - Study ‘groups’ and the EnTT library

Implementing Behaviour Trees in C++

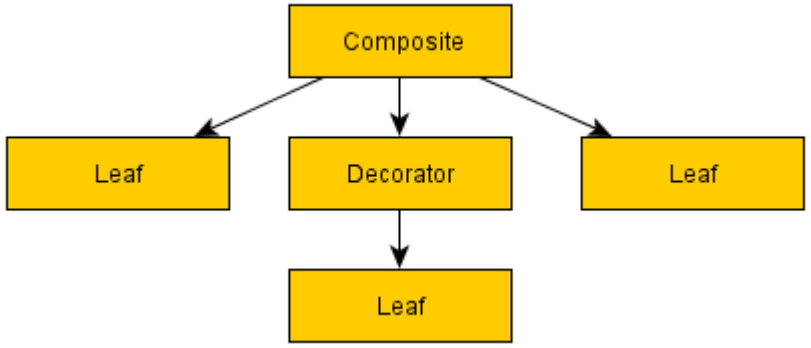
++

A nice visual representation



But how can we code this?

Node Types – a single interface




```
enum class BTState
{ Running, Success, Failure };

class BTNode {
public:
    virtual ~BTNode() noexcept = default;
    virtual void init(ECS::Entity e) {};
    virtual BTState process(ECS::Entity e) = 0;
};
```

Note, the ~BTNode() destructor definition as **`virtual`** *is needed for proper deletion-through-pointer of classes inheriting from BTNode.*

Self study: Why the virtual destructor?

by Tim Straubinger

```
struct Base {
   ~Base(){ std::cout << "~Base()\n";
};
struct Derived : Base {
   ~Derived(){ std::cout << "~Derived()\n";
}
int main(){
   Base* p = new Derived();
   delete p;
   return 0;
}
```

Because `~Base` is not `virtual`, the expression `delete p` where `p` is a `Base*` has no way of knowing how to call `~Derived` instead. This causes Undefined Behaviour, although in practice, you'll typically just see a call to `~Base()` only and anything in `Derived()` gets leaked. As-is, the above code will most likely output just `~Base()`. If you add `virtual` to the `Derived` destructor, you should see the output `~Derived()` \ `~Base()` instead.

A natural question would be "but if I add any virtual function, why is the destructor not automatically made virtual???“

- Somebody in the C++ standard messed it up ☺

Leaf Nodes - turn around

Functionality

- ***init(…)***
  - *Called by parent to initialize*
  - *Not called gain before returning Success/Failure*

- ***process(…)***
  - *Called every frame/tick the node is running*
  - *Does internal processing, interacts with the world*
  - *Returns Running/Success/Failure*

```
class Turn : public BTNode
{
  void init(ECS::Entity e) {
  }

  BTstate process(ECS::Entity e) {
    // modify world
    auto& vel = ECS::registry<Motion>.get(e).velocity;
    vel = -vel;

    // return progress
    return BTState::Success;
  }
};
```

Leaf Nodes - run 3 steps

Functionality

- *init(…)*
  - *Called by parent to initialize*
  - *Not called gain before returning Success/Failure*
- *process(…)*
  - *Called every frame/tick the node is running*
  - *Does internal processing, interacts with the world*
  - *Returns Running/Success/Failure*

```
class RunNSteps : public BTNode
public:
    RunNSteps(int steps) noexcept
        : m_targetSteps(steps), m_stepsRemaining(0) {}
private:
    void init(ECS::Entity e) override {
       m_stepsRemaining = m_targetSteps;
    }

    BTState process(ECS::Entity e) override {
      // update internal state
      --m_stepsRemaining;

      // modify world
      auto& motion = ECS::registry<Motion>.get(e);
      motion.position += motion.velocity;

      // return progress
      if (m_stepsRemaining > 0)
        return BTState::Running;
      else
        return BTState::Success;
    }

    int m_targetSteps;
    int m_stepsRemaining;
};

// Instanciate dynamically
std::shared_ptr <BTNode> run3 = std::make_unique<RunNSteps>(3);
```

Recap: Shared Pointers

- *Issue 1: We can only store objects of the same type in a std::vector*
  - Different objects would need different space

-> need to store list of pointers to base class (here BTNode)

- *Issue2: Objects on the stack aren't permanent*
  - Exiting the current function deletes objects (lifetime ends)
- *Issue 3: Objects on the heap (new …) are difficult to handle*

-> Use `std::make_unique()` and `std::shared_ptr`

  - *Objects are deleted when all its pointers are deleted*

```
// run and turn leaf nodes
std::shared_ptr <BTNode> run3 = std::make_unique<RunNSteps>(3);
std::shared_ptr <BTNode> turn = std::make_unique<TurnAround>();
std::shared_ptr <BTNode> run1 = std::make_unique<RunNSteps>(1);

// entire tree, starting at the root
auto list = std::vector<std::shared_ptr <BTNode>>({ run3, turn, run1 }));
```

Demo

Decorators - Conditions

```
class BTIfCondition : public BTNode
{
  std::shared_ptr<BTNode> m_child;
  std::function<bool(ECS::Entity)> m_condition;
public:
  BTIfCondition(std::shared_ptr<BTNode> child, std::function<bool(ECS::Entity)> condition)
    : m_child(std::move(child)), m_condition(condition){}

  virtual void init(ECS::Entity e) override {
    m_child->init(e);}

  virtual BTState process(ECS::Entity e) override {
    if (m_condition(e))
      return m_child->process(e);
    else
      return BTState::Success;
  }
};
```

Instantiation

```
BTNode standing = BTIfCondition(child_ptr, [](ECS::Entity e) {return ECS::registry<Motion>.get(e).velocity == 0;})
```

AND Sequences

```
class BTSequence : public BTnode
{
    std::map<ECS::Entity, int> n;
    std::vector< std::shared_ptr<BTnode>> children;
public:
    BTSequence(std::vector< std::shared_ptr<BTnode>> children)
    {
        this->children = children;
    }

    virtual void init(ECS::Entity e)
    {
        n[e] = 0;
        this->children[n[e]]->init(e);
    }

    virtual BTstate process(ECS::Entity e)
    {
        BTstate state = this->children[n[e]]->process(e);
        if (state == BTstate::Failure)
            return BTstate::Failure;
        else if (state == BTstate::Running)
            return BTstate::Running;
        else // (state == BTstate::Success)
        {
            n[e]++;
            if (n[e] >= this->children.size())
                return BTstate::Success;
            else
            {
                this->children[n[e]]->init(e);
                return BTstate::Running;
            }
        }
    }
};
```

- Iterate through children until end or until child returns Failure
- Similar to ‘and’ in ‘if(child[0] && child[1] && …)’
  - Expressions following the first ‘false’ will be ignored

- Further useful composites:
  - Repeat N times
  - Repeat indefinitely
  - Negate **Success/Failure**
  - OR Sequence
  - If … else
  - Exit condition

  - What else???

Leaf Nodes – Generic Version

How can we apply the same BT on different entities?

- **How to store internal states?**
  - *store the state for every entity*
  - *use an std::map*

Minor addition to ECS::Entity

```
// Comparator to use as key in std::map
bool operator <(const Entity& rhs) const
{
    return id < rhs.id;
}
```

```
class RunThreeMeters : public BTNode
{
  std::map<ECS::Entity, int> n;
  void init(ECS::Entity e) {
    n[e] = 3;
  }

  BTState process(ECS::Entity e) {
    // update internal state
    n[e]--;

    // modify world
    ECS::registry<Motion>.get(e).position
    += ECS::registry<Motion>.get(e).velocity;

    // return progress
    if (n[e] > 0)
      return BTState::Running;
    else
      return BTState::Success;
    }
};
```

ECS solves every problem?

***E*ntity**

***C*omponent**

***S*ystem**

When not to use ECS?

- *When information is not shared across Systems*
- ***AND** ECS does not fit naturally*
  - multiple components of the same type associated to the same entity
    - *previous slide: multiple class instances store the same information type in a different context*
  - Entities and Components are still be useful locally
    - *Storing Components in ECS instead of locally is equally performant. Use ECS whenever possible!*
    - *The unique Entity ID can still be useful to associate local information to a global entity!*

```
std::map<ECS::Entity, int> n;
void init(ECS::Entity e) {
    n[e] = 3;
}
```

OpenGL

Camera motion?

Instanced rendering

```
// Draw pebbles using instancing
void Pebbles::draw(const mat3& projection)
{
  // Setting shaders
  glUseProgram(effect.program);

  // Enabling alpha channel for textures
  glEnable(GL_BLEND); glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  glEnable(GL_DEPTH_TEST);

  // Getting uniform locations
  GLint projection_uloc = glGetUniformLocation(effect.program, "projection");
  glUniformMatrix3fv(projection_uloc, 1, GL_FALSE, (float*)&projection);

  // Draw the screen texture on the geometry
  // Setting vertices
  glBindBuffer(GL_ARRAY_BUFFER, mesh.vbo);

  // Mesh vertex positions
  // Bind to attribute 0 (in_position) as in the vertex shader
  glEnableVertexAttribArray(0);
  glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);
  glVertexAttribDivisor(0, 0);

  // Load up pebbles into buffer
  glBindBuffer(GL_ARRAY_BUFFER, m_instance_vbo);
  glBufferData(GL_ARRAY_BUFFER, m_pebbles.size() * sizeof(Pebble), m_pebbles.data(), GL_DYNAMIC_DRAW);

  // Pebble translations
  // Bind to attribute 1 (in_translate) as in vertex shader
  glEnableVertexAttribArray(1);
  glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, sizeof(Pebble),
                        (GLvoid*)offsetof(Pebble, position));
  glVertexAttribDivisor(1, 1);

  // Pebble radii
  // Bind to attribute 2 (in_scale) as in vertex shader
  glEnableVertexAttribArray(2);
  glVertexAttribPointer(2, 1, GL_FLOAT, GL_FALSE, sizeof(Pebble),
                        (GLvoid*)offsetof(Pebble, radius));
  glVertexAttribDivisor(2, 1);

  // Draw using instancing
  // https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glDrawArraysInstanced.xhtml
  glDrawArraysInstanced(GL_TRIANGLES, 0, NUM_SEGMENTS*3, m_pebbles.size());

  // Reset divisor
  glVertexAttribDivisor(1, 0);
  glVertexAttribDivisor(2, 0);
}
```

```
struct Pebble {
  vec2 position;
  vec2 velocity;
  float radius;
};
std::vector<Pebble> m_pebbles;
```

https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glDrawArraysInstanced.xhtml

Instanced Rendering - Init

```
bool Pebbles::init()
{
  std::vector<GLfloat> screen_vertex_buffer_data;
  constexpr float z = -0.1;
  // vertex positions
  for (int i = 0; i < NUM_SEGMENTS; i++) {
    screen_vertex_buffer_data.push_back(std::cos(M_PI * 2.0 * float(i) / (float)NUM_SEGMENTS));
    screen_vertex_buffer_data.push_back(std::sin(M_PI * 2.0 * float(i) / (float)NUM_SEGMENTS));
    screen_vertex_buffer_data.push_back(z);

    screen_vertex_buffer_data.push_back(std::cos(M_PI * 2.0 * float(i + 1) / (float)NUM_SEGMENTS));
    screen_vertex_buffer_data.push_back(std::sin(M_PI * 2.0 * float(i + 1) / (float)NUM_SEGMENTS));
    screen_vertex_buffer_data.push_back(z);

    screen_vertex_buffer_data.push_back(0);
    screen_vertex_buffer_data.push_back(0);
    screen_vertex_buffer_data.push_back(z);
  }

  // Vertex Buffer creation
  glGenBuffers(1, &mesh.vbo);
  glBindBuffer(GL_ARRAY_BUFFER, mesh.vbo);
  glBufferData(GL_ARRAY_BUFFER, screen_vertex_buffer_data.size()*sizeof(GLfloat), screen_vertex_buffer_data.data(), GL_STATIC_DRAW);
  glGenBuffers(1, &m_instance_vbo);
  glBindBuffer(GL_ARRAY_BUFFER, m_instance_vbo);

  // Loading shaders
  if (!effect.load_from_file(shader_path("pebble.vs.glsl"), shader_path("pebble.fs.glsl")))
    return false;

  return true;
}
```