Our game and engine is written from the ground up with moddability in focus, and the AI is no different. This presentation goes over how we do that!

This presentation

- ★ About a game that is in progress.
- ★ About how our AI Entity Component Systems are set up:
  - Blackboards, Abilities, Sensors, Utility Actions, Behaviors, Production chains!
- ★ About how pathfinding and movement works.

The game is still WIP and a few things in this presentation are a bit speculative.
The presentation has two parts. First we look into how we manage entities, which entity systems we have and how they interact. After that, we look at our solution for movement and pathfinding.

Who am I?

Anders Elfgren

anders@warpzonestudios.com

@srekel

Plus these "indie" projects:

- Islands at War (2010)
- Bouncy 2 (2010)
- Cake – Character Animation Kickass Editor (2009-2010)
- Button Runner (2009-2010)
- The Selfish Bird Breeder (2009)
- Ragnarök (2009)
- Candy Time (2008)
- Robots in Saucers (2008)
- Bouncy (2007)
- Ragu (2007)
- Math Jumper (2007)
- Steam Empire (2006)
- City Connect (2006)
- SwecPower (2005)
- A Violent World (2005)
- Poppiluss (2005)
- Village Simulator (2004)
- Pocket Task Force (2001 - 2010)

The game at a glance

2.5D - Command a Dwarven mining colony - Indirect control

So, like...

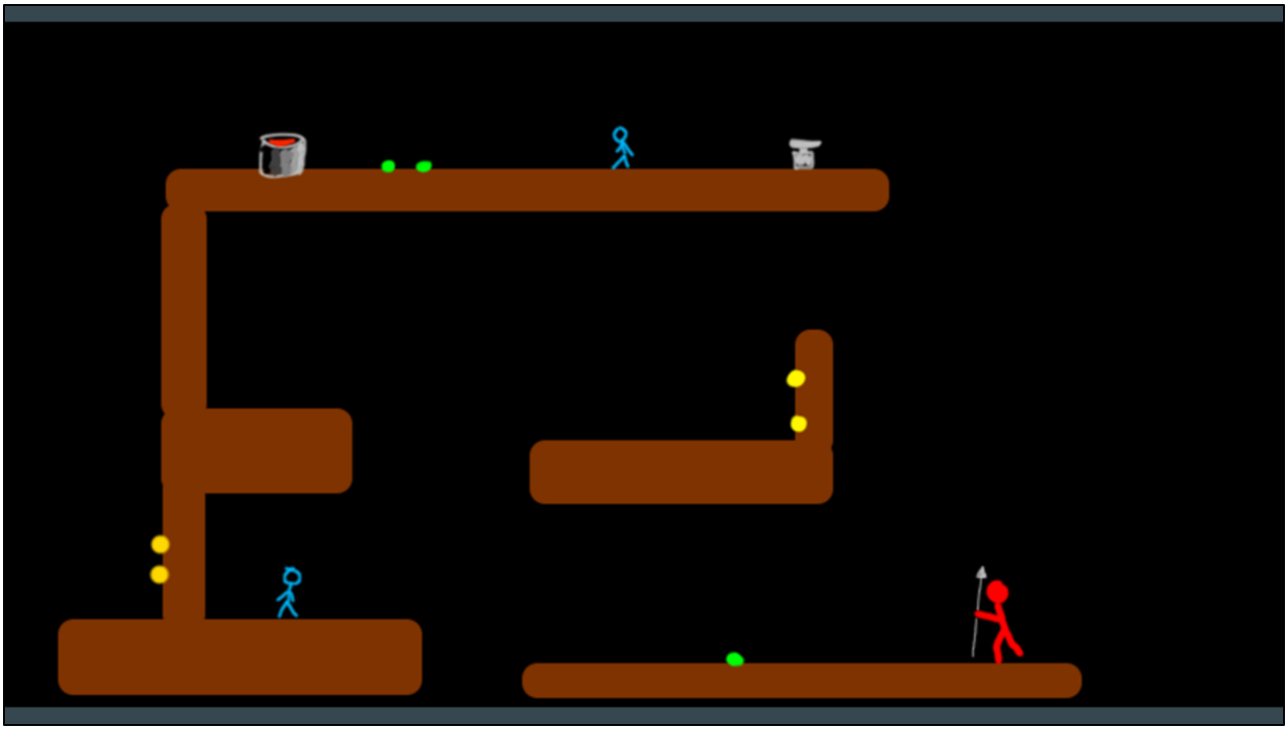
...Terraria, but no directly controllable character.

...Dwarf Fortress, but more casual, and more game than simulation.

...RimWorld, but more game than story-generator.

...Factorio, but more casual.

The game has a platformer perspective - slightly from above.
You control a colony of dwarves *indirectly*, so you place “orders” in the world and if you’re lucky, the dwarves will choose to do that. The dwarves have personal desires that they will want to fulfill in addition to the player’s orders.



A very basic example of the game:
Two blue dwarves that need to decide what to do.
Dig gold, haul green dots, go to a building and craft something, or fight off the invading red troll.
Depending on their skills and items they have many different options available to them.
For example, a dwarf might be only able to walk to the end of a platform and then jump to the next, whereas a more agile dwarf could sprint the last part and jump farther. Or a dwarf could have a grappling hook or hookshot.
This means that we generally can’t reuse path finding between dwarves (not to mention monsters).
If the dwarf in the bottom left chooses to mine, it should pick the gold in the middle of the map, not the gold right next to him. So we need to do a path finding query to find the real distance before we actually choose an action.

The AI at a glance

- ★ Platformer-like, agent-specific traversal.
- ★ Destructible tile-based terrain.
- ★ Make decisions based on player's orders as well as dwarf's personality.
- ★ Mod-friendly, extensible.
- ★ Allow for fine-tuned behavior.

With "fine tuned behavior", I mean it should be possible to design something like a special/unique attack, perhaps one that's enabled when there are a group of enemies in front of the agent, and the attack is a swift move forward that pierces three enemies, and then moves back again to the agent's original position.

The tech at a glance

- ★ Custom engine built in C/C++
- ★ Fixed-point math, fixed timestep
- ★ Entity Component Systems
- ★ Moddability and extensibility through these mechanisms:
  - Compile chain included in game.
  - C APIs

Both engine code and entity management, including all systems, expose their functionality via C APIs.
Some of the C APIs have C++ wrappers for usability. We “eat our own dog food” - use the same APIs we will expose to modders.
Fixed point gives us determinism and universal precision.
We run the game at 10hz.
Cpp files in mods are handled as any other resource: compiled and inserted into our data archiving system so that we can load and execute their hooks at runtime.

Part 1: Entity Management & AI Systems

Entities, Components & Systems

- ★ Entity is 16 bit UID, [1-65535]
- ★ Prefer hashed string UID "tags" to enums

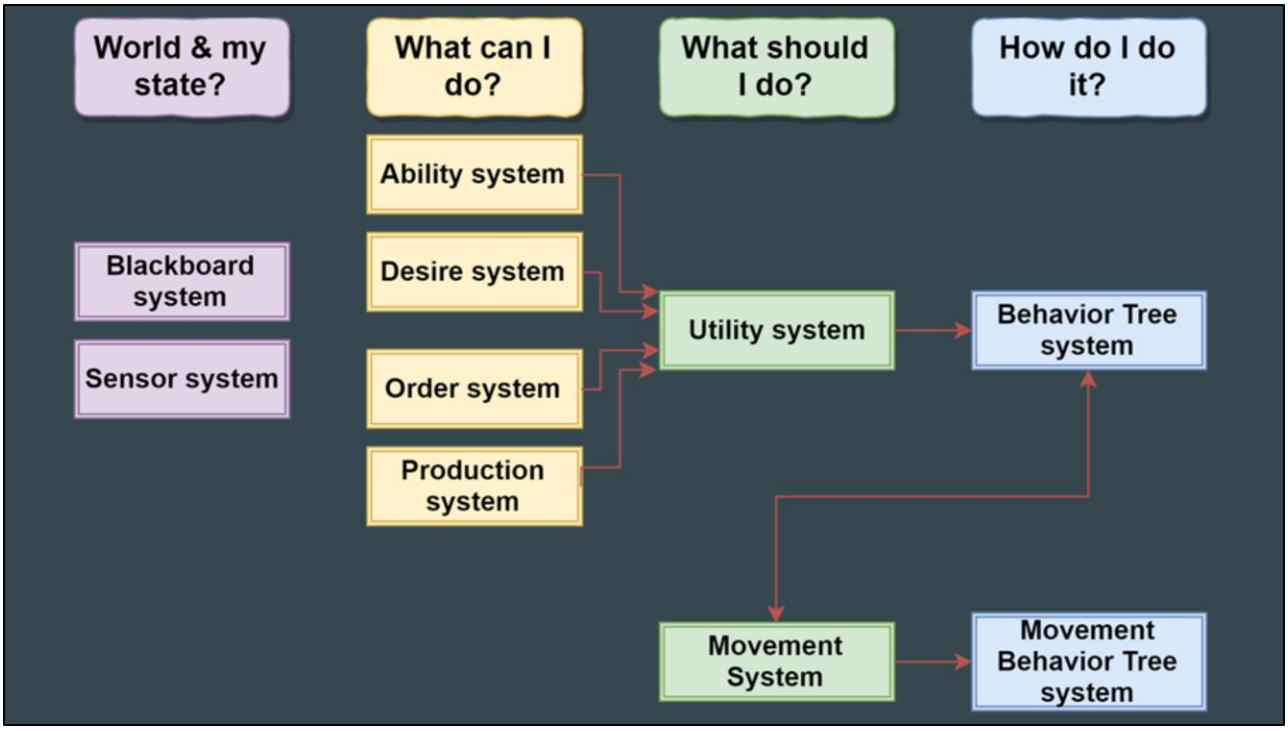
```
entity_component_system sensor_system_create( gamestate* gs );

void sensor_system_destroy( entity_component_system* ecs );
void sensor_system_register_sensors( entity_component_system* ecs,
                                     const sys_u16*            ids,
                                     ai_sensor_func_t*         funcs,
                                     sys_u64                   num_sensors );

void sensor_system_set_sensor_enabled( entity_component_system* ecs,
                                       entity                    entity,
                                       sys_u16                   sensor_id,
                                       bool                      enabled );
```

Entities are simple 16 bit IDs, 1-65535 (ID zero means invalid)
AI systems generally require registering things by an UID rather than accessing by a predetermined enum. Even though it's technically possible to extend enums, UIDs that map to other things via internal system lookups seems like a better solution.
Example of the API for the sensor system in its current state.
Tag UIDs are currently 16 bit but we'll likely change that to something that has less chance of collision.

```
const char* const MODEL_NAME = "dwarf";
const char* const ANIM_NAME  = "idle";
{
  // Spawn dwarf 1
  gamevec3 gamepos( 40, 76, 0 );
  vec3     scale( 1, 1, 1 );

  component_def mydwarf_component_init_data[] = {
    { FIXED_POSITION_COMP,
    { SCALE_COMP, sizeof(

    { MOVEMENT_COMP, 0, nu
    { ABILITY_COMP, 0, nul
    { BLACKBOARD_COMP, 0,
    { AI_SENSOR_COMP, 0, n
    { ORDER_TAKER_COMP, 0,
    { UTILITY_COMP, 0, nul
    { AI_BEHAVIOR_COMP, 0,
    { AI_BEHAVIOR_MOVEMENT_

    { INVENTORY_COMP, 0, n
    { GAME_SYNC_COMP, 0, n
    make_dynamic_transform_
    make_model2d_comp( allocator, MODEL_NAME ),
    make_anim2d_comp( allocator, MODEL_NAME, ANIM_NAME ),
  };
```

```
entity_def mydwarf = { mydwarf_component_init_data,
  static_cast<u16>( wc::arr_len( mydwarf_component_init_data ) ) };
entity ids[1];
em.create_entities( &mydwarf, ids, 1, 0 );

utility_system_add_action(
  &_utility_system, idle_create_action( _gamestate, ids[0] ) );
utility_system_add_action(
  &_utility_system, haul_create_action( _gamestate, ids[0] ) );

sensor_system_set_sensor_enabled(
  &_sensor_system, ids[0], SENSOR_HAUL, true );
```

Create a component definition - a list of components.
Note gamevec3 for _game_ transform - fixed point, and vec3 for _graphics_ (floating point).
From the component definition, create an entity definition.
Create entity and get ID back, store that ID or pass it to additional setup-functions.
Here we have already registered the sensor SENSOR_HAUL and now that we create the entity we ensure that the entity uses it.



This is divided into two main parts, the “mid level” AI decision making and execution, and the movement.
Sensor (or perception) is fed which sensors to run, and generally feeds information back through the blackboard. We have a neat design for the blackboard that I can’t get into now, but we have just a few big allocations instead of one per blackboard (or one per item per blackboard!).
The order and desire system adds and removes available actions to agents. Purpose of order system is to manage adding actions to any agents that can handle carrying out the order.
Production system handles craftable resources, production buildings, and informs one of the actions that are always available - hauling.
Utility system is used for *decision making*, BT system used for *executing*. This allows us to have very small and reusable behavior trees. For example BT_HAUL is just five nodes (Sequence [Move, Pick up, Move, Place]). This is very nice because in my experience, behavior trees that are big, make a lot of decisions, and/or needs to remember its state become unwieldy.
It also means we can reuse behavior trees easily - we could have a robot (or an animal) that could run the same haul BT as the dwarves.
Each Utility action generally just has one purpose, to set up the blackboard for the BT’s execution, and to tell the BT system which BT to run. For BT_HAUL this would be “Move here, pick up this entity, move there, place the entity”.

Why Utility System and Behavior Trees?

- ★ Planner - Powerful but complex to implement and tweak, maybe not right gameplay?
- ★ Behavior Trees - Good for designing a specific course of actions but bad at decision making.
- ★ Utility Systems - Excel at decision making, but doesn't execute.

My first thought was actually to implement GOAP because it seemed like a natural fit. When I actually started on the project, I read up on GOAP and found STRIPS and HTN and started a prototype implementation and realized it would require a lot of work and tweaking to get it right. It seems like this is feeling is common in the game AI community.
I've also realized that the gameplay might not be what we want - the longer the "chain", the more of a simulation and less of a game it becomes. Not to mention less dynamic - what are the chances that the right action will be the same in a minute or two as it is now?

Previously I've worked with using Behavior Trees for decision making and while it worked really well in some ways, there were a lot of issues once the trees grew past a certain size, started having more complex behavior, tried to do proper decision making, and needed to jump back and forth between nodes a lot.

At around this point I started reading up on Utility Systems and it seemed like they are good at the thing that BTs *aren't* good at, and vice versa. It still does. :)

Utility System

Inspired by Infinity Axis Utility System (Dave Mark)

- ★ Actions
- ★ Considerations
- ★ Scoring functions
- ★ Response curves

We also use Dual Utility Reasoning (Kevin Dill, GAIP2) for prioritizing actions.

Problem:

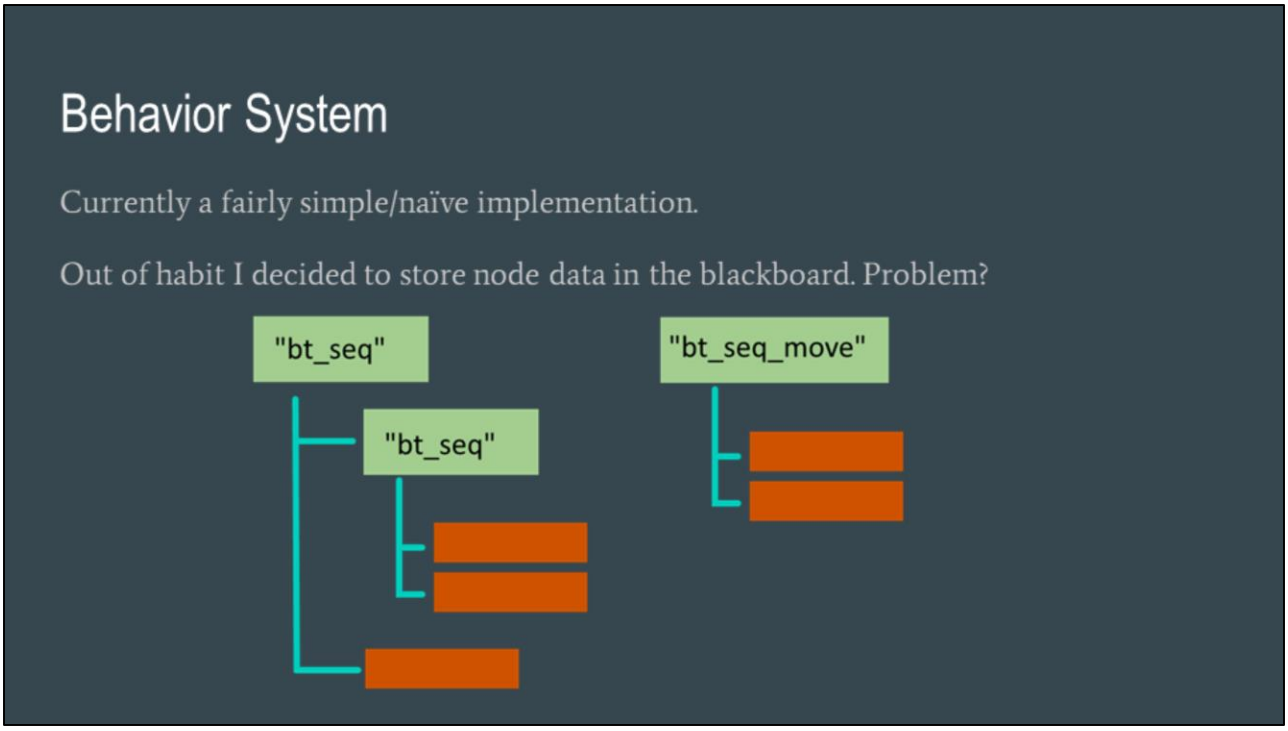
Considerations!

- ★ Fast execution.
- ★ Consistent execution time.
- ★ Should not have side-effects.

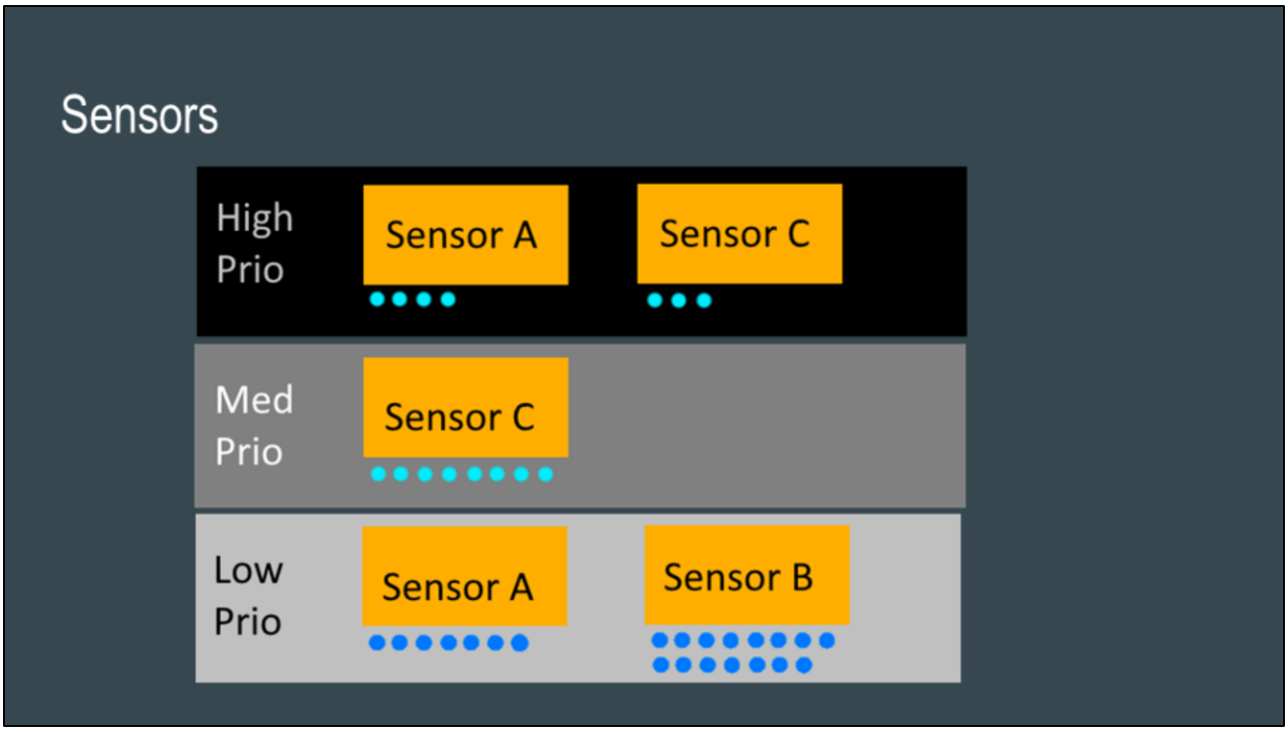
Solution: Sensors

Considerations are [0-1], and they don't look at each other to create good scores, which means it can be hard to get good scoring for different categories, or priorities, of actions.
One way is to artificially boost certain actions, which we may do in the future, but for now, we have implemented DUR. Maybe jumping the gun, but seems like a good approach.
Kevin Dill's DUR in GAIP2:
http://www.gameaipro.com/GameAIPro2/GameAIPro2_Chapter03_Dual-Utility_Reasoning.pdf

Another problem with a utility system is that, since there may be so many actions and you need to evaluate all of the considerations for an action to figure out its score, the considerations must be consistently fast. It also is reasonable to expect them to be pure and not have side effects.
This is tricky if they do all the calculations. For example "Get me the best resource on the map to haul to a building". Expensive, and if the consideration doesn't store which one it chose, then the action or behavior needs to re-do the calculation.
We offload this type of work to sensors. Sensors can be expensive and are easier to load balance.



Our BT implementation is fairly basic - bordering on naïve - but that's fine because our BT logic should be fairly simple - do this, then that.
Since utility system handles decisions we don't need as many conditions. This could change but for now - fine.
Problem: BT nodes check their config value from the blackboard. If you have two nodes with the same ID, they'll use the same memory. This is manageable when defining a single tree because you can see the IDs of all nodes there, but when you have two separate BTs running at the same time, it can be harder to spot the problem. We don't have a good solution for this right now.
We can splice in a sub-tree easily enough and is likely something we'll use to be able to "data drive" certain behaviors, but of course that also creates the same double-id problem.
BTs are used not only for characters but also for buildings and other things that need to follow a certain script.



I don't think we do anything particularly new with sensors, nor are we very far along in the implementation (though we have one or two running), so this is slightly speculative.
The idea is to be able to tell the sensor system which sensor your agent is interested in, and how important it is. High prio = run every frame, medium prio = run at least every X frame, low prio = run when there's time over, basically.
As mentioned, sensors can be quite expensive since we can slice up their updates quite freely.
They are also fairly dumb - they do some calculations and then store the result in the blackboard, ready to look at by both considerations, actions, and behaviors.

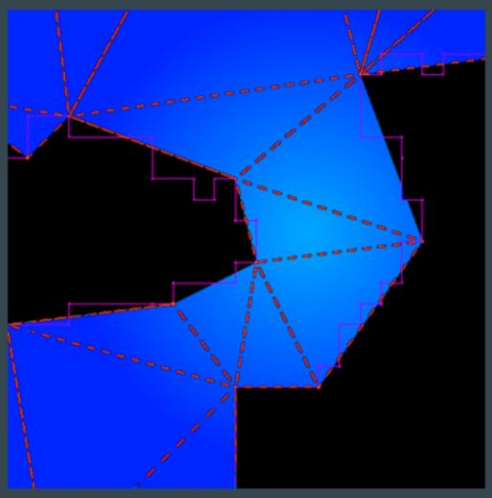
Part 2: Movement

Navigation: Requirements

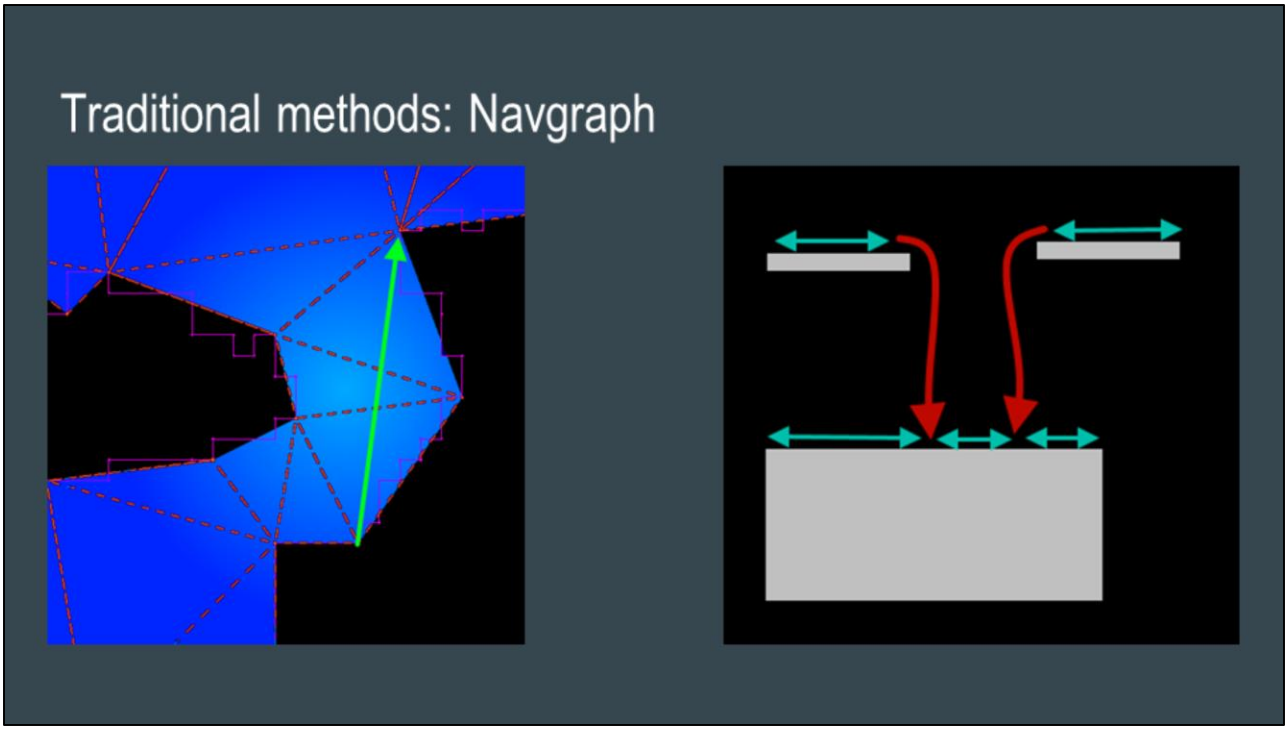
- ★ Tile based world
- ★ Agents with different styles and options of movement
- ★ Expose everything to modders
- ★ Dynamic world
  - Tiles can change.
  - Paths can become invalid.
  - Agents need to be able to follow invalid paths.

We want them to be able to follow invalid paths because we don't want to 1) regenerate navigation data and 2) re-evaluate all affected paths instantly whenever the terrain changes.

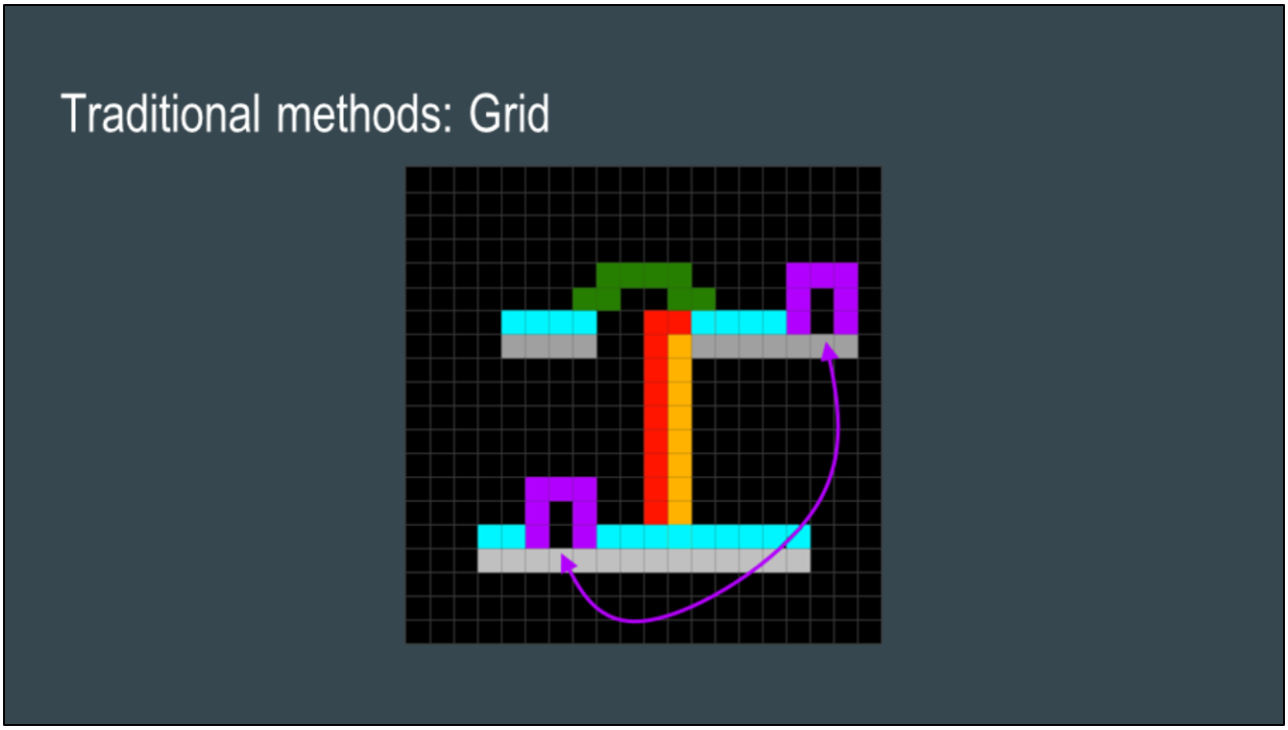
Traditional methods: Navmesh



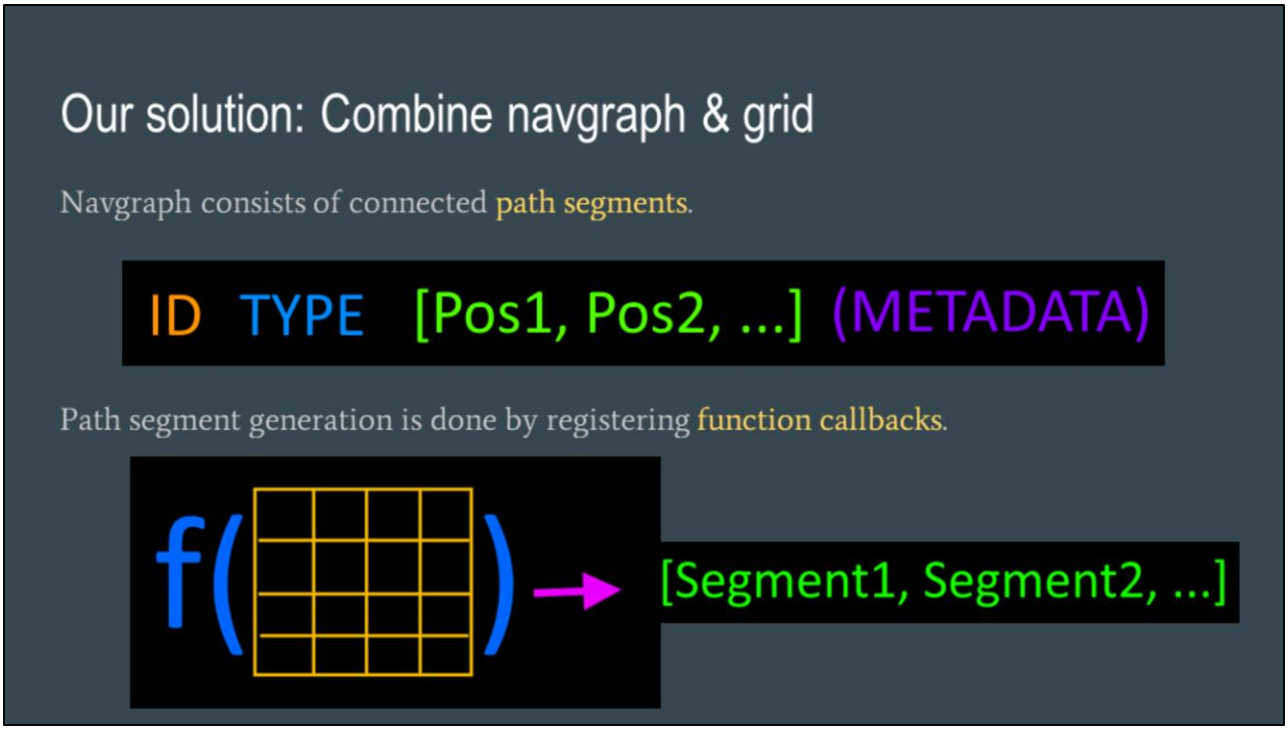
Navmeshes only really work well for when moving along the X axis and the Y axis are the same. This is not the case for our game so we don't use it.
Image taken from https://gamedev.stackexchange.com/questions/38721/how-can-i-generate-a-navigation-mesh-for-a-tile-grid



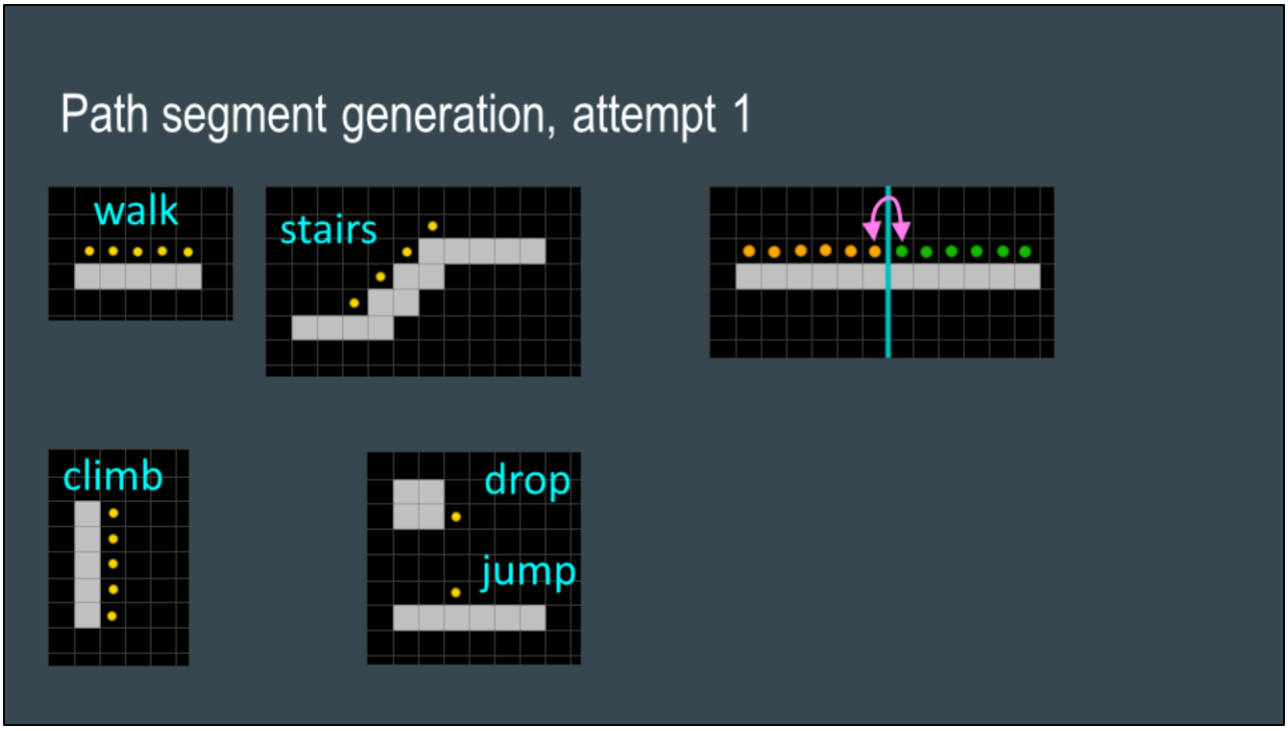
But maybe you could generate the mesh and instead move along the lines instead of the triangles?
Problem: You may need to generate many more edges - for example the green one in the left image.
Another problem with navgraphs is that edges may need to be cut up many many times, causing lots of nodes.



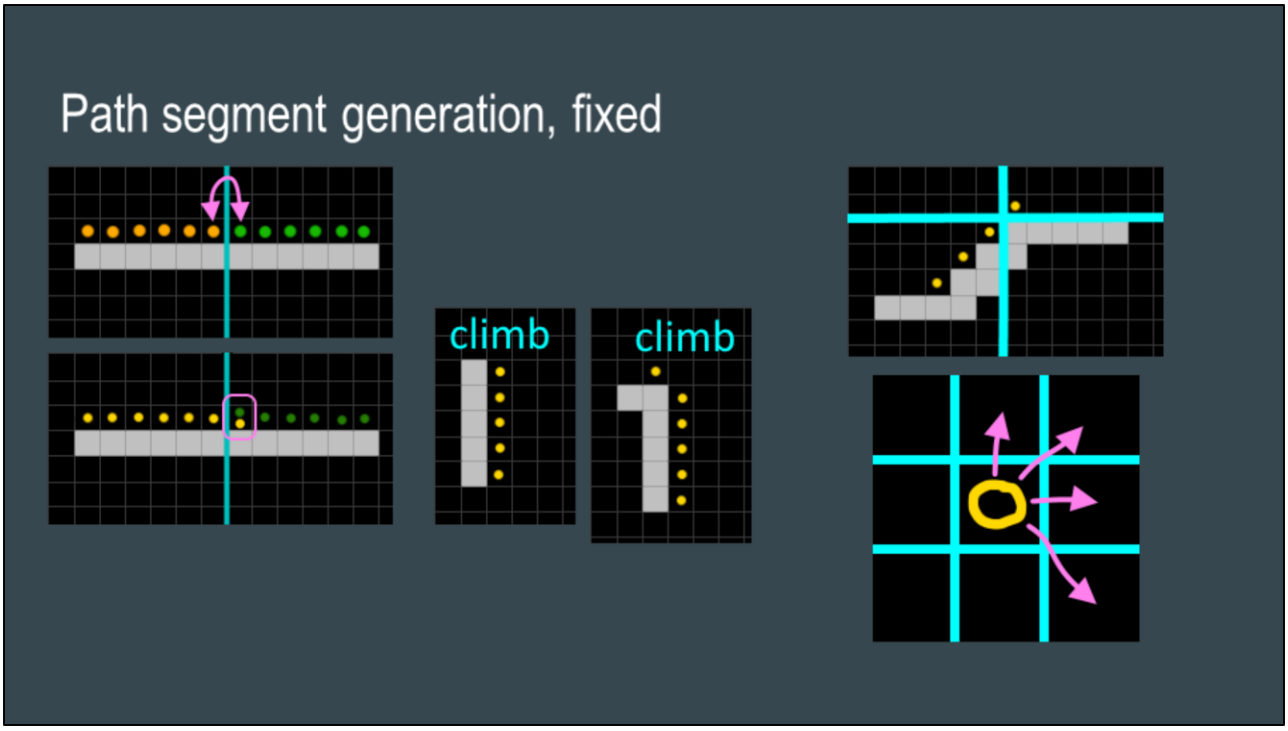
Ok, so can we do a traditional A* along a grid?
Doesn’t work well for teleporters - we need another solution *on top of* the grid solution for that (e.g. smart objects)
Tricky to do uni-directional paths (jump down).
Tricky to do the kind of natural movement we want.



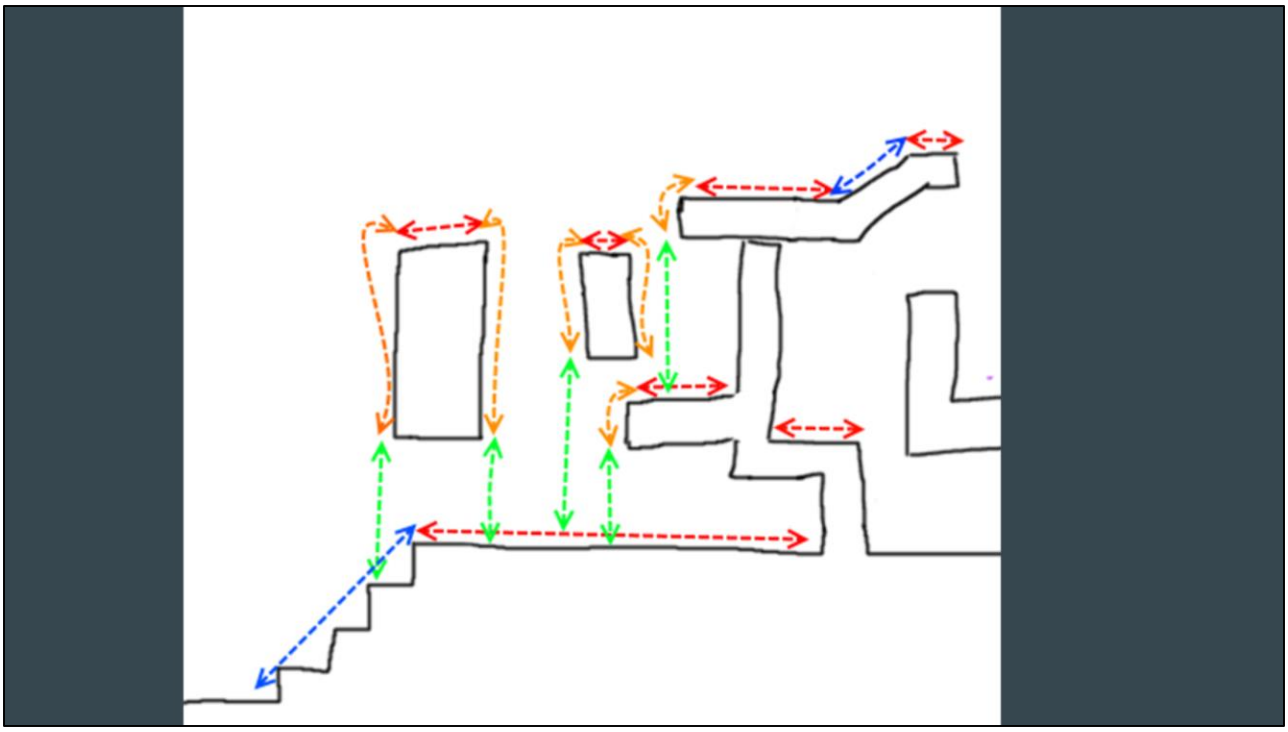
A path segment has an ID, a type, a list of grid positions, and optional metadata.
Metadata can be something like “how far is it to the ceiling in this walk path”.
Each block is regenerated in one swoop - all registered functions called on the area.
Function takes a 16 x 16 area (block) and returns a list of path segments in that area.



Walk path is simple: Find a tile that has *something* and the tile above is *nothing*. Then keep going right until that's not true anymore or you reach the end of the block. Drop and jump up-paths don't have intermittent points because we don't want to be able to connect to a different path in the middle of a jump.
Problem: How do you connect two path segments that are in neighboring blocks? You could theoretically do some ad hoc "well my next is also a WALK type segment so I'll just continue to walk there" but how does that work when the segment types are different?



Solution: Path segments need to *extend* into the next block.
Segments connect IFF they overlap at at least one position.
Path generation functions need to be look at the four indicated neighbors.
I'm pretty sure that's good enough - i.e. I assume that a block doesn't need to go left because whichever path segment is there, is supposed to have gone right.
This means that when a block is dirtied, we need to re-generate it AND its eight neighbors. Also note that path segments can extend further than one step into neighboring block.



This is an example of what a bunch of generated path segments could look like (ignoring block borders)

Navigation: Path finding

We use A*.

First attempt: Store the paths' IDs. Didn't work when terrain changed.

Solution: Store full path segments.

Stored paths are continuously validated by a low-prio sensor.

Couldn't store IDs and later assume we could look them up, because if a block was regenerated, the IDs might be invalid.
We keep the full list of path segments as a single buffer.

Navigation: Path following

When running A*, each path segment is checked against the agent's abilities.

Each ability can say "I can / can't traverse this path type", and if yes, it will return the cost of doing so.

Abilities map to a behavior tree ID. When the path is followed and a new segment is encountered, the agent's movement behavior tree component switches to that behavior tree.

The behavior tree is responsible for deciding when to switch to the next path segment.

Recap!

This talk has covered the following topics:

- ★ Utility system for decision making -- BTs for execution.
- ★ Agent-specific path-finding and movement in dynamic platformer-style 2D.
- ★ Abilities to define agents and make them unique.
- ★ Prioritized sensors to control performance and handle complicated Utility Considerations.

Questions?

•••

@srekel
/u/srekel
anders@warpzonestudios.com
We have a Discord! Join us :)