Kruger Entity Model

Bobby Anguelov

- Kruger (KRG) is an Experimental/Prototype Engine
- Named after the Kruger national park in South Africa
- Started in 2011
- Will never be finished…

Caveats

- The KRG entity model is a prototype for how I think such a system should work
- It is nothing fancy or new…
- It's a hybrid of the ECS model and the GameObject-Component model.
- It is definitely not production ready…

Goals

- Simple to understand
- Simple to author
- Entities need to be updateable atomically
- Entity updates need to be trivially parallelizable

Entity

- Entities are objects
- Entities contain an array of components
- Entities contain an array of systems

- Entities (can) have updates
- Entities have an explicit spatial hierarchy
- Dependencies between entities are explicitly defined

Entity Component

- Data Storage
  - Components have a list of properties (reflected members)
  - Reflection system auto-generates serialization and resource loading code
- Primary serialization and streaming mechanism
- Components have no access to other components or to the entity
- Components do not have an update
- Inheritance of components is allowed

Entity Component

- Components can also contain logic and can perform operations on their data

- E.G. Animation Graph Component
  - All graph update logic lives on the component.
  - We have methods to update the graph and get the resulting pose

- Each component is a STANDALONE black box
  - No dependencies to other components' logic or data is allowed

Entity Component

Two main types:

- Entity Component
  - Empty component
  - Name
  - UUID
- Spatial Entity Component
  - Derives from Entity Component
  - Has a local transform
  - Has local bounds (OBB)

Spatial Entity Components

- Has a local transform and local bounds (OBB)
- Has a world transform and world bounds (OBB)
  - Inaccessible to derived classes
- Has a parent spatial component plus attachment socket ID
  - Inaccessible to derived classes
- Has a list of child spatial components
  - Inaccessible to derived classes

Spatial Entity Components

When a local transform is updated, the world transform is updated:

The world transform update is as follows:

1. The world transform is recalculated based on the parent component
2. The world bounds are updated based on the parent component
3. All children are asked to update their world transforms

Spatial Entity Components

When a local transform is updated, the world transform is updated:

The world transform update is as follows:

- The world transform is recalculated based on the parent component
- The world bounds are updated based on the parent component
- All children are asked to update their world transforms

This means world transforms are always accurate!

Spatial Entity Components

Additional Notes:

- Bounds are not inclusive
  - Bounds only refer to the individual components
  - Can't think of a valid use case for inclusive bounds
- Setting a local transform can be expensive for long hierarchy chains
  - We rarely have deep hierarchy chains
  - We very rarely update transforms multiple times per frame

Entity Component – Spatial Hierarchy

- Each entity has a single root spatial component

- If this spatial root component is set, the entity is considered a spatial entity and so has a position in the world.

Entity Component Access

- There is no access to components via the entity
- Components have no access to their entity
- Having such access is the biggest problem with the game-object component model…
  - Create hidden dependencies between components
  - Creates circular dependencies between components
  - Any entity can access the internals of any other entity which is a parallelization nightmare.

Local Entity Systems

- Each entity can have a set of systems
- An entity system is a local system
  - It has an update
  - It can only operate on its parent entity's components
  - Can have transient runtime state
- Entity systems are responsible for updating components and for data transfer between components

Local Entity Systems

- Local entity systems are updated via the entity
- Conceptually we “update” the entity but in fact we just update all local systems for that entity
  - There is no actual entity update

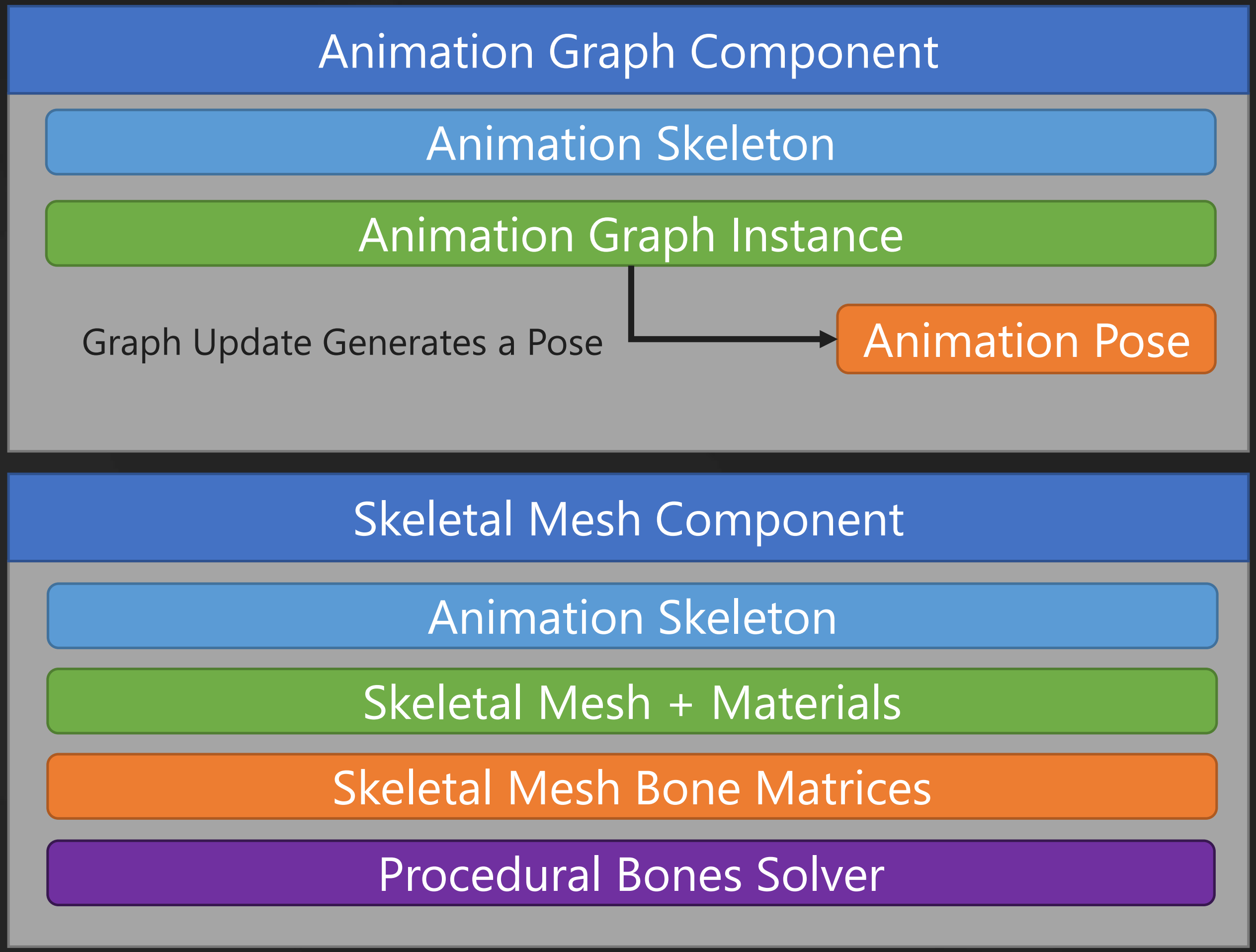
Local Entity System Example

Animation System

Components:
- Animation Component
- Skeletal Mesh Component

Runtime State:
- Bone mapping table between anim skeleton and mesh skeleton

Update:
1. Updates animation Component
   - *This generates a pose*
2. Transfers pose to skeletal mesh component
   - *Using bone mapping table*
3. Updates procedural deformation bones on skeletal mesh component



Global Entity Systems

We also have global entity systems (These are more traditional ECS style systems)

- These are singleton systems
  - One instance per entity world
- They operate on a set of components
  - Based on component type
- They have an update
- They can have transient runtime state

Global Entity System Example

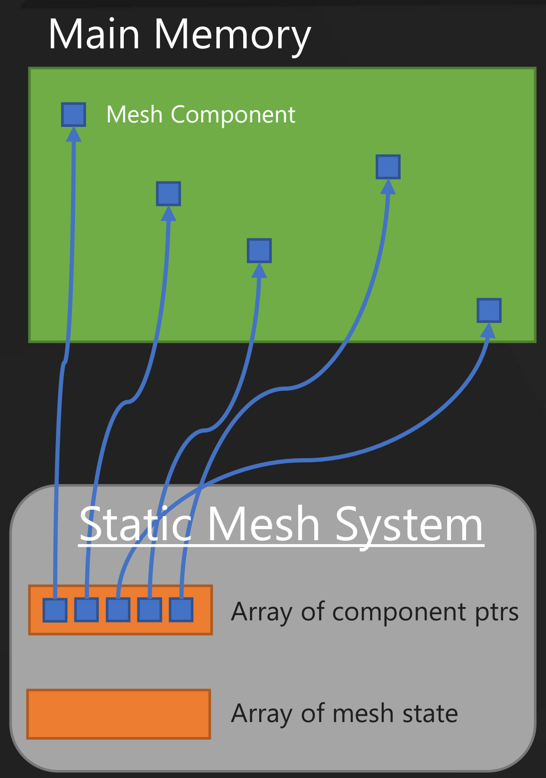
Static Mesh System

- All static mesh components are registered with the system
- Upon registration the system maintains two additional data structures
  - AABB BVH for all fixed (immobile) static meshes
  - Flat array for all mobile static meshes
- Once per frame does broadphase culling of visible meshes and submits to renderer

Global Entity Systems

- Important to note that global systems don't need to operate on components directly
- Components are used as information that a certain thing exists in the game world i.e. there is a static mesh and it has a position
- Systems can allocate efficient internal state based on their specific needs
- That state can be injected into components for use during entity updates

Global Entity System Example



- Static mesh components are all over main memory
- Iterating over them would be slow and inefficient
- So when registering them with the static mesh system, we allocate internal storage and reflect the relevant mesh component information into it
- The system does not actually use the components and only ever references its allocated runtime state memory
  - We trade off more memory for improved performance since we often duplicate some component state e.g. world transforms, bounds, etc…
- This gives system full control of data layout allowing programmers to optimize the layout per system and per use-case
- Systems can also allocate runtime state for components
  - E.g. pose storage, physics actors, etc…

Entity World

- There is a world in which all entity exist
- The world has an update
- That update is broken up into fixed stages
  - Start-Frame, Pre-Physics, Physics, Post-Physics, End-Frame
- Per update stage
  1. Update all entities
  2. Update all global entity systems

Updates

- Local and Global Systems specify which stages they need to be updated in
- When registering for a stage update, they also specify a priority for that stage e.g. <**Stage**: Pre-Physics, **Priority**: 65>
- This allows fine grain control of update order between systems without creating actual dependencies between them

Entity Dependencies

- Entities are not allowed to directly or depend on other entities apart from a spatial dependency
  - All other entity dependencies are expected to be inaccurate or frame-lagged e.g. targeting, reading an entity positions, etc…
- An entity may request that it be attached to another entity very much like spatial components are attached.
  - We do not allow circular dependencies for attachments (obviously)
- This will result in the parent entity being scheduled for update before the children
  - This allows the parent entity to update its transform and other data before the child, thereby ensuring spatial coherency

Entity Memory Layout

- Since entities are treated as atomic units, they are allocated as such.

- Each entity is allocated in a single block of memory

- Memory Layout:
  - First the entity
  - Followed by the entity systems
  - Finally all the components ordered by type

- This means dynamic component creation is a special case.
  - Either new components are allocated on the heap and we just use ptrs to them
  - Or we move the whole entity each time a component is added (expensive and complex)
  - *In my experience though, dynamic creation of components is not something that's really needed once you have a sensible entity type generator and/or entity templates, so we don't support this in KRG.*

Parallelism

- KRG has little to no global state
- Since entity updates are atomic per update stage, we parallelize all entity updates
- If there are spatial dependencies, we create an update chain and update the parent immediately followed by children on the same thread.
- Global systems are updated on the main thread and are expected to parallelize their work internally as needed.

Questions?

http://twitter.com/bobby_anguelov