Visit me on the Web:
http://www.TerranceCohen.com

Follow me on Twitter:
http://twitter.com/terrance_cohen

Connect with me on LinkedIn:
http://www.linkedin.com/in/terrancecohen

Ask me a question:
http://www.formspring.me/terrancecohen
At a high level, I have three sections to this discussion

1. First, we'll talk about, "Why are we here?"
   1. Not, "Why are we here on this earth," that's outside the scope of this presentation. I mean, "What do I hope you'll conclude from this."
      o We'll discuss the problem we're trying to solve, and my approach to a solution
2. Second, we'll talk about details of my solution, which I call the Dynamic Component System.
   o We'll discuss features of the system, and why it's a good solution
3. And third, we'll dig into implementation details of the system,
   o Using a few systems of dynamic components for illustration purposes
First we'll talk about the problem we're trying to solve.
Purpose: Statement of Problem

- Monolithic / deep Game Object hierarchy
  - Memory: binds data @ compile time
    - Allocated throughout lifetime

- This is the traditional OO game object based gameplay model

- Member data allocated throughout lifetime, even when not in use
  - Some designs will allocate some data separately from the game object, e.g. a physics instance
  - But this is counter to the architecture, rather than central to the architecture,
  - And each system is likely to roll it’s own solution, since the unifying architecture doesn’t support it
<table>
<thead>
<tr>
<th>Purpose: Statement of Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Monolithic / deep Game Object hierarchy:</td>
</tr>
<tr>
<td>o Memory: binds data @ compile time</td>
</tr>
<tr>
<td>o <strong>Performance: poor cache coherence</strong></td>
</tr>
<tr>
<td>▪ Arrays of non-homogeneous objects</td>
</tr>
<tr>
<td>▪ Virtual dispatch</td>
</tr>
<tr>
<td>▪ Fragmented instance data</td>
</tr>
</tbody>
</table>

| • Encourages iteration over arrays of non-homogeneous objects |
| • Encourages virtual dispatch in the most unpredictable patterns |
| • Instance data fragmented by unused elements |
| • These are general issues with performance under OO |
Capabilities of an object are fixed at compile time, fully determined by class.

Changing capabilities means changing hierarchy often impossible without code duplication or multiple inheritance of implementation.

Both have serious drawbacks.

Changing capabilities through multiple inheritance:
- Some will argue that there are no serious drawbacks to MI
- Beyond the scope of this discussion
- Even for strong proponents of such architecture, should agree that it's not a good solution to this problem
  - Messy unification of disparate hierarchies
Purpose : Statement of Problem

- Monolithic / deep Game Object hierarchy:
  - Memory: binds data @ compile time
  - Performance: poor cache coherence
  - Architecture: capability <-> inheritance

- "What we're used to"

- But select the best tool for the job.

- "What we're used to," so easy to overlook flaws
- Step back. Select the best tool for the job.
Purpose : Statement of Problem

- Monolithic / deep Game Object hierarchy:
  - Memory: binds data @ compile time
  - Performance: poor cache coherence
  - Architecture: capability <-> inheritance

- "What we're used to"

- But select the best tool for the job.

- There's a better way!
So here's my attempt at a solution. It's worked well for us. I hope it'll work well for you.
Purpose: Proposed Solution

- Construction of Game Object through composition of components at runtime
Purpose : Proposed Solution

- Construction of Game Object through composition of components at runtime
- Simple!
- Thank you for coming!
Purpose : Proposed Solution

- Construction of Game Object through composition of components at runtime
- Simple!
- Thank you for coming!
  - Oh, you want details !?!
Everything that the Game Object represents can be broken-down into small chunks
Each component represents a data transformation
Ultimately, that's all the game is doing
  o e.g. state machine + states

e.g.
  • model
  • animation
  • physics
  • ai/logic
  • sensing
  • send/receive messaging
  • causing damage
So now we'll talk about my solution which I call the Dynamic Component System.

First we'll talk at a high level about what is the system...
The Dynamic Component System

- Evolution
  - Resistance 1 - Proof of Concept (1 type)
  - Resistance 2 - "Early Adopters" (32 types)
  - Ongoing (295 types as of May 1st 2010)
    - Majority of new gameplay code
    - Large portions of old gameplay refactored
The Dynamic Component System

- Evolution
  - Resistance 1 - Proof of Concept (1 type)
  - Resistance 2 - "Early Adopters" (32 types)
  - Ongoing (295 types as of May 1st 2010)
    - Majority of new gameplay code
    - Large portions of old gameplay refactored

- So it’s mature.
The Dynamic Component System

- Evolution
  - Resistance 1 - Proof of Concept (1 type)
  - Resistance 2 - “Early Adopters” (32 types)
  - Ongoing (295 types as of May 1st 2010)
    - Majority of new gameplay code
    - Large portions of old gameplay refactored

- So it’s mature. (No, not *that* way.)

Not in the “adults only” sense. It’s been exercised.
The Dynamic Component System

- Side-by-side implementation
  - Not necessary to refactor existing code
  - Coexist with components
- General solution

- Side-by-side implementation
  - Not necessary to refactor existing code
  - Game objects in the traditional (or "legacy") monolithic model can coexist and interoperate with components
- The solution is general - doesn't require a particular game architecture.
  - Some environment-specific implementation details
The Dynamic Component System

• Does not address matters of
  o Reflection
  o Serialization
  o Data building
  o Instance versioning
  o … those things are handled separately
    ▪ outside the scope of this discussion
Now we'll discuss features of the system, a.k.a. why I think it's a good solution.
Dynamic Component System: Features

- Components
- High-performance
- Dynamic
- System
Dynamic Component System: Features

- **Components**
  - Originally called Aspects from AOP
  - Derived from base Component class
    - 8 bytes of administrative data
  - Allocated from pools
    - One pool per concrete type
    - "Roster" indexes instances
    - "Partition" separates allocated/free instances
High-performance

• All operations, except for "search", are small constant-time
  o Allocate/free
  o Resolve handle
  o Get type
  o Type implements (derived from)
• No instance copying
  o manipulations are done on the Roster rather than the instance pool
Dynamic Component System: Features

- Components
- High-performance
  - Updates per-type (per-pool)
    - Cache friendly
  - Encourage async update
    - e.g. on SPU
      - Roster: contiguous list of alloc'd instances
      - Roster up to partition is DMA list

High-performance
- Updates are performed per-type (per-pool)
  - Cache friendly
- Designed to encourage & simplify async update
  - e.g. on SPU
    - Roster contains contiguous list of pool indices of allocated instances
    - Roster up to partition is DMA list
High-performance:

- References between components normally handled by holding/resolving handles

Dynamic Component System: Features

- Components
- **High-performance**
  - Resolving handle
    - Small, constant-time operation:
      - Index into Pool
      - Compare generation
      - Return Component*
Dynamic Component System: Features

- Components
- High-performance
- **Dynamic**
  - Runtime composition of game objects
    - Dynamically alter behavior without baggage
  - Component allocated == in use
  - Pool sizes == max concurrent allocations

Dynamic:
- Runtime composition of game objects
  - Objects can substantially alter any aspect without carrying around any "baggage" from other states
- Recommended idiom: component is allocated IFF actively in use
- Pool sizes need only accommodate max concurrent allocations
Dynamic Component System: Features

- Components
- High-performance
- **Dynamic**
  - High-frequency alloc() & free()
    - alloc():
      - test for availability
      - make handle from index & generation
      - increment Roster Partition
      - Component::Init()

Dynamic:
- Results:
  - High-frequency alloc() & free()
    - Small constant-time operations:
      - alloc():
        - test for availability
        - make handle from index & generation
        - increment roster partition
        - call Init on allocated component
Dynamic Component System : Features

- Components
- High-performance
- **Dynamic**
  - High-frequency `alloc()` & `free()`
    - `alloc()`:
    - `free()`:
      - `Component::Deinit()`
      - swap Roster index with Partition-adjacent index
      - decrement Partition
      - increment generation

Dynamic
- Results:
  - High-frequency `alloc()` & `free()`
  - Small constant-time operations:
    - `free()`:
      - call Deinit
      - swap index in roster with partition-adjacent index
      - decrement partition
      - increment generation

So let's take an example to put some of these pieces together.

We'll look at an example of freeing a dynamic component instance.
Here's the prototype for DynamicComponent::Free(). It takes...

So here's an example.

We have a pool of instances of some given component type.

We have a roster, which is an array of indices into the instance pool. (Note that all of the instances in the pool are of the same type, so they are of equal size. So the value at a roster index is itself just an index into the pool.

You can see that we have a partition value, which separates roster indices representing allocated vs. free instances.
Alright. Now we're going to free the component represented by the 3rd roster element, which is currently index 3 in the pool.
So what we'll do is swap the 3rd and 4th roster elements.
Now we're freeing the 4th roster element, which represents the 3rd instance in the pool.

Since we swapped the roster entry for the element to be freed into the partition-adjacent roster entry, all we have to do to free that instance is...
... decrement the partition.

And voila, we've free'd a component instance with a few very small constant-time operations, and we didn't touch the instance data, only indices.
Dynamic Component System: Features

- Components
- High-performance
- Dynamic
- **System**
  - Not all-or-nothing!
  - Examples:
    - Conversation
    - Script Events
    - Shots: no game object

System
- Doesn't need to completely replace a standard game object model
- Can completely replace standard game object model
  - Other systems host components
    - Conversation
    - Script Events
  - Shots
    - No game object
• At a high level, I have three sections to this discussion

1. So now we're digging into implementation details of the system
   - APIs always a good place to start
namespace DynamicComponent
{
  //
  // Hosts' API
  //
  // Call these from GameObjects (or other objects) that host Dynamic Components
  //

  // Allocate a component of type, add it to host's component chain,
  // and optionally park a pivot in the component
  // returns null if no space is available for allocation
  Component* Allocate (Type type, HostHandle host_handle,
                         Chain* chain, void* pivot = NULL);
namespace DynamicComponent
{
    //
    // Hosts' API
    //
    // Call these from GameObjects (or other objects) that host Dynamic Components
    //
    Component* Allocate ( Type type, HostHandle host_handle,
                           Chain* chain, void* prius = NULL );

    // resolve a ComponentHandle to a Component
    // returns NULL if component_handle is null or is a stale handle
    // (i.e. component instance has been reused)
    Component* ResolveHandle ( Type type, ComponentHandle component_handle );
namespace DynamicComponent
{
  //
  // Hosts' API
  // Call these from GameObjects (or other objects) that host Dynamic Components
  //
  Component* Allocate ( Type type, HostHandle host_handle,
                         Chain* chain,
                         void* pcius = NULL );
  Component* ResolveHandle ( Type type, ComponentHandle component_handle );

  // get one component of type that belongs to host
  Component* Get ( Type type, HostHandle host_handle, Chain chain );
/*
 * Hosts' API
 * 
 * Call these from GameObjects (or other objects) that host Dynamic Components
 */

namespace DynamicComponent
{
    // Type type, HostHandle host_handle,
    // Chain* chain, void** prius = NULL;
    Component* Allocate
    {
    }

    // Type type, ComponentHandle component_handle;
    Component* ResolveHandle
    {
    }

    // Type type, HostHandle host_handle, Chain chain;
    Component* Get
    {
    }

    // Get the first Component in host's chain that implements the type interface
    Component* GetComponentThatImplements
    {
    }
}
namespace DynamicComponent
{
    //
    // Hosts' API
    //
    // Call these from GameObjects (or other objects) that host Dynamic Components
    //
    Component* Allocate (Type type, HostHandle host_handle,
                          Chain* chain, void** ptrs = NULL);
    Component* ResolveHandle (Type type, ComponentHandle component_handle);
    Component* Get (Type type, HostHandle host_handle, Chain chain);
    Component* GetComponentThatImplements (Type type, HostHandle host_handle, Chain chain);

    // Get all Components of type in host's chain, up to a max of count instances.
    // count should be passed with the size of the component array.
    // On return, count will contain the number of matching components, up to the specified limit.
    Component** GetComponents (Type type, HostHandle host_handle,
                               Chain chain, uint32 count);
}
namespace DynamicComponent
{
    //
    // Hosts' API
    //
    // Call these from GameObjects (or other objects) that host Dynamic Components
    //
    Component* Allocate ( Type type, HostHandle host_handle, Chain* chain = NULL );
    Component* ResolveHandle ( Type type, ComponentHandle component_handle );
    Component* Get ( Type type, HostHandle host_handle, Chain chain );
    Component** GetComponentThatImplements ( Type type, HostHandle host_handle, Chain chain );
    Component** GetComponentsThatImplement ( Type type, HostHandle host_handle, Chain chain, u32 count );

    // get all Components in host's chain that implement type's interface.
    // count should be passed with the size of the component array.
    // on return, count will contain the number of matching components, up to the specified limit.
    Component** GetComponentsThatImplement ( Type type, HostHandle host_handle, Chain chain, u32 count );
}
namespace DynamicComponent
{
  //
  // Hosts' API
  //
  // Call these from GameObjects (or other objects) that host Dynamic Components
  //
  Component* Allocate
  { Type type, HostHandle host_handle, Chain* chain, void** prius = NULL; }
  Component* ResolveHandle
  { Type type, ComponentHandle component_handle; }
  Component* Get
  { Type type, HostHandle host_handle, Chain chain; }
  Component** GetComponentThatImplements
  { Type type, HostHandle host_handle, Chain chain; }
  Component** GetComponents
  { Type type, HostHandle host_handle, Chain chain, u32 count; }
  Component** GetComponentThatImplements
  { Type type, HostHandle host_handle, Chain chain, u32 count; }

  //free the component from host's component chain
  void Free
  { Type type, HostHandle host_handle, Chain& chain, ComponentHandle& component_handle; }
}
namespace DynamicComponent
{
    //
    // Hosts' API
    //
    // Call these from GameObjects (or other objects) that host Dynamic Components
    //
    Component* Allocate
    ( Type type, HostHandle host_handle,
      Chain* chain, void* prius = NULL );

    Component* ResolveHandle
    ( Type type, ComponentHandle component_handle );

    Component* Get
    ( Type type, HostHandle host_handle, Chain chain );

    Component* GetComponentThatImplements
    ( Type type, HostHandle host_handle, Chain chain );

    Component** GetComponents
    ( Type type, HostHandle host_handle,
      Chain chain, u32 count );

    Component** GetComponentThatImplements
    ( Type type, HostHandle host_handle,
      Chain chain, u32 count );

    void Free
    ( Type type, HostHandle host_handle,
      Chain chain, ComponentHandle* component_handle );

    //free all of the components in host's component chain
    // (GameObject automatically free their component chain when they are destroyed)
    void FreeChain
    ( HostHandle host_handle, Chain chain );
namespace DynamicComponent
{
    //
    // Hosts' API
    //
    // Call these from GameObjects (or other objects) that host Dynamic Components
    //
    Component* Allocate ( Type type, HostHandle host_handle, Chain* chain, void* ptr = NULL );
    Component* ResolveHandle ( Type type, ComponentHandle component_handle );
    Component* Get ( Type type, HostHandle host_handle, Chain chain );
    Component* GetComponentThatImplements ( Type type, HostHandle host_handle, Chain chain );
    Component** GetComponents ( Type type, HostHandle host_handle, Chain chain, u32 count );
    Component** GetComponentsThatImplements ( Type type, HostHandle host_handle, Chain chain, u32 count );
    void Free ( Type type, HostHandle host_handle, Chain chain, ComponentHandle component_handle );
    void FreeChain ( HostHandle host_handle, Chain chain );

    // downcast a Component* to one of its subclasses.
    // please use this instead of the c-style '(Type*)object' so that casts are checked in debug
    // Example:
    //
    #define COMPONENT_CAST(component, type) \
    (Component*)ValidCast(component, DynamicComponent::type)

    inline Component* ValidCast ( Component* component, Type type );
namespace DynamicComponent
{
  // Host's API
  Component* Allocate ( Type type, HostHandle host_handle, 
                         Chain* chain, void* prius = NULL );
  Component* ResolveHandle ( Type type, ComponentHandle component_handle );
  Component* Get ( Type type, HostHandle host_handle, Chain chain );
  Component* GetComponentThatImplements ( Type type, HostHandle host_handle, 
                                           Chain chain );
  Component** GetComponents ( Type type, HostHandle host_handle, 
                               Chain chain, u32 count );
  Component** GetComponentsThatImplement ( Type type, HostHandle host_handle, 
                                            Chain chain, u32 count );

  void Free ( Type type, HostHandle host_handle, Chain chain, 
              ComponentHandle component_handle );
  void FreeChain ( HostHandle host_handle, Chain chain );

  #define COMPONENT_CAST(component, type) \ 
  ((type) Component**)(component, DynamicComponent::type)

  inline Component* ValidCast ( Component* component, Type type );

  //
  // Systems' API
  //
  // call these from systems that use the u3u

  // get a list of component types that implement the interface of the given component type
  // count should be passed with the size of the types array.
  // on return, count will contain the number of matching component types.
  // up to the specified limit.

  Type* GetTypesThatImplements ( Type type, u32 count );
namespace DynamicComponent
{
    // Hosts' API
    Component* Allocate ( Type type, HostHandle host_handle,
                          Chain* chain, void* prius = NULL );
    Component* ResolveHandle ( Type type, ComponentHandle component_handle );
    Component* Get ( Type type, HostHandle host_handle, Chain chain );
    Component** GetComponentThatImplements ( Type type, HostHandle host_handle, Chain chain );
    Component** GetComponents ( Type type, HostHandle host_handle,
                                 Chain chain, u32 count );
    void GetComponentsThatImplements ( Type type, HostHandle host_handle,
                                      Chain chain, u32 count );
    void FreeChain ( HostHandle host_handle, Chain chain );

    #define COMPONENT_CAST(component, type) \
    ( (type)Component::*ValidCast ( component, DynamicComponent::type ) )
    inline Component* ValidCast ( Component* component, Type type );

    // Systems' API
    //
    // Call these from systems that use the DCS
    //
    Type* GetTypesThatImplements ( Type type, u32 count );

    // returns whether component type implements interface
    bool TypeImplements ( Type type, Type interface );
}
namespace DynamicComponent
{
    // Hosts' API
    Component* Allocate
    { Type type, HostHandle host_handle,
      Chain* chain, void* prius = NULL; }
    Component* ResolveHandle
    { Type type, ComponentHandle component_handle; }
    Component* Get
    { Type type, HostHandle host_handle, Chain chain; }
    Component** GetComponentThatImplements
    { Type type, HostHandle host_handle, Chain chain; }
    Component** GetComponents
    { Type type, HostHandle host_handle,
      Chain chain, u16 count; }
    Component** GetComponentsThatImplements
    { Type type, HostHandle host_handle,
      Chain chain, u32 count; }
    void Free
    { Type type, HostHandle host_handle, Chain chain,
      ComponentHandle component_handle; }
    void FreeChain
    { HostHandle host_handle, Chains chain; }

    #define COMPONENT_CAST(component, type) \ 
    {((Component**)ValidCast(component, DynamicComponent::type))}
    inline Component* ValidCast
    { Component* component, Type type; }

    // Systems' API
    //
    // Call these from systems that use the DCS
    //
    Type* GetTypesThatImplement
    { Type type, u32s count; }
    bool TypeImplements
    { Type type, Type interface; }

    // returns the number of components of type that are currently allocated
    u32 GetNumAllocated
    { Type type; }
namespace DynamicComponent
{
    // Hosts' API
    Component* Allocate ( Type type, HostHandle host_handle,
                          Chain* chain, void* prius = NULL );
    Component* ResolveHandle ( Type type, ComponentHandle component_handle );
    Component* Get ( Type type, HostHandle host_handle, Chain chain );
    Component* GetComponentThatImplements ( Type type, HostHandle host_handle, Chain chain );
    Component** GetComponents ( Type type, HostHandle host_handle,
                                 Chain chain, u32 count );
    Component** GetComponentsThatImplement ( Type type, HostHandle host_handle,
                                              Chain chain, u32 count );
    void Free ( Type type, HostHandle host_handle, Chain chain,
                ComponentHandle component_handle );
    void FreeChain ( HostHandle host_handle, Chains chain );
}

#define COMPONENT_CAST(component, type) \  
    ((type*)Component::*ValidCast (component, DynamicComponent::type))

inline Component* ValidCast ( Component* component, Type type );

    // Systems' API
    //
    // Call these from systems that use the DCS
    //
    Type* GetTypesThatImplement ( Type type, u32 count );
    bool TypeImplements ( Type type, Type interface );
    u32 GetNumAllocated ( Type type );

    //returns an array of pointers to all allocated components of type, their count,
    //and their size
    Component** GetComponents ( Type type, u32 count );
    //returns an array of all components of type (including unallocated instances),
    //an array of the indices of allocated components within that array,
    //and the count of indices
    Component* GetComponentsIndexed ( Type type, u16* indices, u32 count );
namespace DynamicComponent
{
    // Hosts' API
    Component* Allocate ( Type type, HostHandle host_handle,
                            Chain* chain, void* prius = NULL );
    Component* ResolveHandle ( Type type, ComponentHandle component_handle );
    Component* Get ( Type type, HostHandle host_handle, Chain chain );
    Component* GetComponentThatImplements( Type type, HostHandle host_handle, Chain chain );
    Component** GetComponents ( Type type, HostHandle host_handle, Chain chain, u32 count );
    Component** GetComponentsThatImplement ( Type type, HostHandle host_handle,
                                              Chain chain, u32 count );
    void Free ( Type type, HostHandle host_handle, Chain chain, ComponentHandle& component_handle );
    void FreeChain ( HostHandle host_handle, Chain chain );

    #define COMPONENT_CAST(component, type) \
    ((type) Component* Cast(component, DynamicComponent::type))
    inline Component* ValidCast ( Component* component, Type type );

    // Systems' API
    // Call these from systems that use the DCS

    Type* GetTypesThatImplement ( Type type, u32 count );
    bool TypeImplements ( Type type, Type interface );
    u32 GetUnAllocated ( Type type );
    component** u32 GetComponents ( Type type, u32 count );
    Component* GetComponentsIndexed ( Type type, u16*4 indices, u32* count );

    // updates all components of those types that want to be updated in the given UpdateStage
    void UpdateComponents ( UpdateStage::Run stage );
}
namespace DynamicComponent
{
  // Hosts' API

  Component* Allocate
  ( Type type, HostHandle host_handle,
    Chain* chain, void* prius = NULL );

  Component* ResolveHandle
  ( Type type, ComponentHandle component_handle );

  Component* Get
  ( Type type, HostHandle host_handle, Chain chain );

  Component** GetComponents
  ( Type type, HostHandle host_handle, Chain chain,
    Chain chain, u32 count );

  Component** GetComponentsThatImplement
  ( Type type, HostHandle host_handle,
    Chain chain, u32 count );

  ComponentHandle GetComponentThatImplements
  ( Type type, HostHandle host_handle, Chain chain );

  void Free
  ( Type type, HostHandle host_handle, Chain chain,
    ComponentHandle component_handle );

  void FreeChain
  ( HostHandle host_handle, Chains chain );

#define COMPONENT_CAST(component, type) \
  ((type*)Component**)ValidCast(component, DynamicComponent::type)

#line Component* ValidCast
  ( Component* component, Type type );

  // Systems' API

  // Call these from systems that use the DCS

  Type* GetTypesThatImplement
  ( Type type, u32 count );

  bool TypeImplements
  ( Type type, Type interface );

  u32 GetNumAllocated
  ( Type type );

  u32 GetComponents
  ( Type type, u32 count );

  Component** GetComponentsIndexed
  ( Type type, u16 indices, u32 count );

  void UpdateComponents
  ( Type type, u32 count );

  // Frees a component that was allocated without a host, and is not in any chain

  void Free
  ( Type type, ComponentHandle component_handle );
namespace DynamicComponent
{
    
    // Hosts' API
    Component* Allocate ( Type type, HostHandle host_handle, Chain* chain, void* prius = NULL );
    Component* ResolveHandle ( Type type, ComponentHandle component_handle );
    Component* Get ( Type type, HostHandle host_handle, Chain chain );
    Component* GetComponentThatImplements ( Type type, HostHandle host_handle, Chain chain );
    Component** GetComponents ( Type type, HostHandle host_handle, Chain chain, u32* count );
    Component** GetComponentsThatImplement ( Type type, HostHandle host_handle, Chain chain, u32* count );
    void Free ( Type type, HostHandle host_handle, Chain& chain );
    void FreeChain ( HostHandle host_handle, Chains chain );

    #define COMPONENT_CAST(component, type) \    
    ((type*)Component::*ValidCast(component, DynamicComponent::type))

    inline Component* ValidCast ( Component* component, Type type );
    
    // Systems' API
    //
    // Call these from systems that use the DCBS
    //
    Type* GetTypesThatImplement ( Type type, u32* count );
    bool TypeImplements ( Type type, Type interface );
    u32 GetNumAllocated ( Type type );
    Component** vecComponents ( Type type );
    Component* GetComponentsIndexed ( Type type, u16* indices, u32* count );
    void UpdateComponents ( UpdateStage::Enum stage );
    void Free ( Type type, ComponentHandle& component_handle );

    //returns the current PPU UpdateStage::Enum.
    // will be UpdateStage::None unless UpdateComponents() is on the stack.
    UpdateStage::Enum GetCurrentUpdateStage ( );
}
```cpp
namespace DynamicComponent
{

    // Host's API
    Component* Allocate ( Type type, HostHandle host_handle,
                          Chain* chain, void* prius = NULL );
    Component* ResolveHandle ( Type type, ComponentHandle component_handle );
    Component* Get ( Type type, HostHandle host_handle, Chain chain );
    Component** GetComponentThatImplements ( Type type, HostHandle host_handle, Chain chain );
    Component** GetComponents ( Type type, HostHandle host_handle, Chain chain,
                                 Chain chain, u32 count );
    Component** GetComponentsThatImplement ( Type type, HostHandle host_handle,
                                             Chain chain, u32 count );
    void Free ( Type type, HostHandle host_handle, Chain chain,
                ComponentHandle& component_handle );
    void FreeChain ( HostHandle host_handle, Chain chain );

    // System's API
    // Call these from systems that use the DCB

    Type* GetTypesThatImplement ( Type type, u32 count );
    Type* GetImplementedTypedefs ( Type type, Type interface );
    u32 GetNumAllocated ( Type type );
    u32 GetNumComponents ( Type type, u32 count );
    Component* GetComponentsIndexed ( Type type, u16* indices, u32 count );
    void UpdateComponents ( Type stage, u32 count );
    void Free ( Type type, ComponentHandle& component_handle );
    u8 GetCurrentUpdateStage ( );

    // returns true iff type updates in stage
    u8 GetTypeUpdateStages ( Type type );
}
```
namespace DynamicComponent {

// Host's API

Component𝐳 Allocate ( Type type, HostHandle host_handle, 
                      Chain* chain, void* prius = NULL );
Component𝐳 ResolveHandle ( Type type, ComponentHandle component_handle );
Component𝐳 Get ( Type type, HostHandle host_handle, Chain chain );
Component𝐳 GetComponentThatImplements ( Type type, HostHandle host_handle, Chain chain );
Component𝐳 GetComponent ( Type type, HostHandle host_handle, Chain chain, u32 count );
Component𝐳 GetComponentThatImplements ( Type type, HostHandle host_handle, 
                                          Chain chain, u32 count );
void Free ( Type type, HostHandle host_handle, Chain chain, 
            ComponentHandle component_handle );
void FreeChain ( HostHandle host_handle, Chain chain );

#define COMPONENT_CAST(component, type) \ 
    (type#ComponentådeValidCast{component, DynamicComponent::type})

inline Component谔 ValidCast ( Component谔 component, Type谔 type谔 );

// Systems' API

Type谔 GetTypeThatImplements ( Type谔, u32谔 count谔 );
bool TypeImplements ( Type谔, Type谔 interface谔 );
u32 GetUnallocated ( Type谔 );
Component谔 GetComponent ( Type谔, u32谔 count谔 );
Component谔 GetComponentIndexed ( Type谔, u16谔 index谔, u32谔 count谔 );
void UpdateComponents ( UpdateStages谔 enum谔 stage谔 );
void Free ( Type谔, Component谔 handles谔 component谔 handle谔 );
u8 GetTypeUpdateStages ( Type谔 );
}
A Dynamic Component Architecture for High Performance Gameplay

- Purpose

- The Dynamic Component System

- Implementation by Example
  - API
  - **Script Events: Type Registration**
  - Navigation: Allocation & Init
  - Shots: Asynchronous Update
Script Events : Type Registration

- Script Events are Components
  - Possibly related to a Game Object, but
  - Hosted by the Script Event System

- Registered (allocated) from Lua
  - When conditions are met, call back

- Satisfaction:
  1. Updated: poll their conditions
  2. Notified by gameplay

- Allocated on-demand from Lua scripts
  - When specified conditions are met, callback to script

- Two modes of testing for satisfaction:
  1. Some event types are updated automatically by the Dynamic Component System, and poll their conditions
  2. Other event types are notified of occasions during gameplay that may cause them to become satisfied
**Script Events : Type Registration**

<table>
<thead>
<tr>
<th>Notified</th>
<th>Updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Checkpoint</td>
<td>• Location</td>
</tr>
<tr>
<td>• Damage</td>
<td>• Timer</td>
</tr>
<tr>
<td>• Death</td>
<td>• Visible</td>
</tr>
<tr>
<td>• NumAlive</td>
<td></td>
</tr>
<tr>
<td>• Active</td>
<td></td>
</tr>
<tr>
<td>• Custom</td>
<td></td>
</tr>
<tr>
<td>o Reload</td>
<td></td>
</tr>
<tr>
<td>o Zoom</td>
<td></td>
</tr>
<tr>
<td>o Grapple</td>
<td></td>
</tr>
<tr>
<td>o SeatEnter</td>
<td></td>
</tr>
</tbody>
</table>
Types are registered through the `DynamicComponent::TypeRegistrar`. The registrar is responsible for keeping track of:

- the types registered with the system,
- Their base classes for testing which types implement which interfaces
- And during which update stages each type wants to be updated by the DCS
e.g. Say the ScriptEventSystem just has a base type and two concrete event types.

The base class is a component type named ScriptEvent. We'll register two component type that are subclass of ScriptEvent

To illustrate type registration, one will be notified, and one will be an event that updates both on PPU and asynchronously, e.g. on an SPU. So we'll use LocationEvent for example.

- Upon system initialization, it registers both types
- LocationEvents update on the SPU during PreUpdate to poll various conditions for satisfaction. It'll stash the results back in the component
- LocationEvents also update on the PPU during PostUpdate for an opportunity to invoke script callbacks if they're satisfied
At a high level, I have three sections to this discussion:

1. First, we'll talk about our purpose here.
   - We'll discuss the problem we're trying to solve, and my approach to a solution
2. Second, we'll talk about details of my solution, which I call the Dynamic Component System.
   - We'll discuss features of the system, and why it's a good solution
3. And third, we'll dig into implementation details of the system,
   - Using a few systems of dynamic components for illustration purposes
Navigation : Allocation & Initialization

- NavComponents are hosted by Game Objects
- Updated by the Navigation system

- NavComponents are hosted by Game Objects (usually Bots) to query the navigation database (mesh + dynamic obstructors)
- Updated by the Navigation system, not automatically by the Dynamic Component System
  - For navigation-specific load balancing
- Interesting initialization idiom
Navigation: Allocation & Initialization

- Remember the API call to allocate a component:

```c
// allocate a component of type, add it to host's component chain,
// and optionally park a prius in the component
// returns NULL if no space is available for allocation
Component* DynamicComponent::Allocate( Type type, HostHandle host_handle,
                                        Chain* chain, void* prius = NULL );
```

- WTF is a prius?
  
  In Dynamic Component parlance, a prius is, "A lightweight vehicle for transporting initialization data."
  
  Opportunity to pass runtime or design-time data to a component instance.

- Prius is simply passed to the initialization method of a component instance upon allocation.

- But is that safe!?!
  
  Component must cast to an assumed type!
Navigation: Allocation & Initialization

- Remember the API call to allocate a component:

```c
// allocate a component of type, add it to host's component chain,
// and optionally park a prius in the component
// returns NULL if no space is available for allocation
Component* DynamicComponent::Allocate( Type type, HostHandle host_handle,
                                        Chain* chain, void* prius = NULL );
```

- WTF is a prius?
  - initialization data
  - Runtime or design-time data

- void* But is that safe!?

- WTF is a prius?
  - In Dynamic Component parlance, a prius is, "A lightweight vehicle for transporting initialization data."
  - Opportunity to pass runtime or design-time data to a component instance.

- Pirus is simply passed to the initialization method of a component instance upon allocation.

- But is that safe!?!?
  - Component must cast to an assumed type!
• Initialization
  o NavQuery is the Prius for a NavComponent
  o NavQuery::Submit() allocates a NavComponent, and passes itself as the prius.

```cpp
NavComponent* NavQuery::Submit(GameObject* host)
{
  DynamicComponent::Type type = GetComponentType(); // virtual member of NavQuery
  dynamicComponent::Component* component = host->AllocateComponent(type, this);
  NavComponent* nav = COMPONENT_CAST(component, Nav);
  return nav;
}
```

• host->AllocateComponent()? Helper in GameObject:

```cpp
DynamicComponentComponent*
GameObject::AllocateComponent(DynamicComponent::Type type, void* prius)
{
  return DynamicComponent::Allocate(type, m_handle, &m_component_chain, prius);
}
```

• Interesting initialization idiom
  o A NavQuery is the Prius for a NavComponent
  o NavQuery::Submit() allocates a NavComponent, and passes itself as the prius.
Navigation: Allocation & Initialization

• Gameplay code example

```cpp
void FollowerHot::Initialize(GameObject* target) {
    Nav::GameObjectQuery source_query [this]; //query closest point to game object
    Nav::GameObjectQuery target_query  [target];
    Nav::PathQuery  path_query   [source_query, target_query];
    Nav::PathComponent* n_path   = path_query.Submit());
}
```

• When navigation components are updated, endpoints and path are dynamically recomputed on SPU

Obviously highly simplified, but meant to convey the point that that a Nav::Query is a prius, whose Submit() method returns the result of DynamicComponent::Allocate(), passing itself as void* prius.
• At a high level, I have three sections to this discussion

1. First, we'll talk about our purpose here.
   o We'll discuss the problem we're trying to solve, and my approach to a solution

2. Second, we'll talk about details of my solution, which I call the Dynamic Component System.
   o We'll discuss features of the system, and why it's a good solution

3. And third, we'll dig into implementation details of the system,
   o Using a few systems of dynamic components for illustration purposes
Hosted by Game Object
Replaces Projectile GameObjects

Two DynamicComponent Type hierarchies:
1. Shot represents a state machine
   - Notified
2. ShotAction represents the state of a Shot
   - Updated
   - Shared

Hosted by Game Object that fired the Shot
Completely replace the Projectile type of GameObject
Two DynamicComponent Type hierarchies:
1. Shot represents a state machine
   - Don't update
   - They are notified of events by their current ShotAction(s)
2. ShotActions represent the states of a Shot
   - Shared among different Shot types
   - Automatically & asynchronously updated by the Dynamic Component System
The ShotMoveForward component type has different declarations on the PPU and on an SPU.

On the PPU, it is a polymorphic type, derived from ShotAction.

On the SPU, however, it is a POD type.

This allows us to operate on instances in an object-oriented way on the PPU, without a need to fix-up vtables in a different address space.

Note that m_location is the first member of ShotMoveForward on SPU, whereas m_next_location is a member of the base class ShotAction on the PPU.
"Sentinel" data member identifies start of SPU instance
  o This is achieved by placing a "sentinel" in the base class declaration identifying where an instance of the class begins when being DMA’d to an SPU.
  o The sentinel is a zero-sized array member
What we've discussed

1. First, we talked about our purpose here.
   - We discussed the problem we were trying to solve, and my approach to a solution

2. Second, we talked about details of my solution, which I call the Dynamic Component System.
   - We discussed features of the system, and why it's a good solution

3. And third, we dug into implementation details of the system,
   - Using a few systems of dynamic components for illustration purposes
What we've discussed

1. First, we talked about our purpose here.
   - We discussed the problem we were trying to solve, and my approach to a solution
2. Second, we talked about details of my solution, which I call the Dynamic Component System.
   - We discussed features of the system, and why it's a good solution
3. And third, we dug into implementation details of the system,
   - Using a few systems of dynamic components for illustration purposes
What we've discussed

1. First, we talked about our purpose here.
   o We discussed the problem we were trying to solve, and my approach to a solution
2. Second, we talked about details of my solution, which I call the Dynamic Component System.
   o We discussed features of the system, and why it's a good solution
3. And third, we dug into implementation details of the system,
   o Using a few systems of dynamic components for illustration purposes
Closing thoughts.

Tricky aspect: components that share data or otherwise communicate
2 options:

1. Select one to own the data, other references through owner by resolving handle
   1. E.g. ShotActions call into Shot to notify of events
2. Scatter-gather idiom
   1. PU update stages gather inputs for & scatter outputs from SPU processes
   2. Involves copying
   3. Better for async processing