A Dynamic Component Architecture for High Performance Gameplay

Terrance Cohen
Lead Systems Engineer
Insomniac Games

Game Developers Conference™ Canada
May 6-7, 2010
Vancouver Convention Centre | Vancouver, BC
www.GDC-Canada.com

UBM
REBOOT
INSOMNIAC GAMES
GDC Canada

Visit me on the Web:

http://www.TerranceCohen.com

Follow me on Twitter:

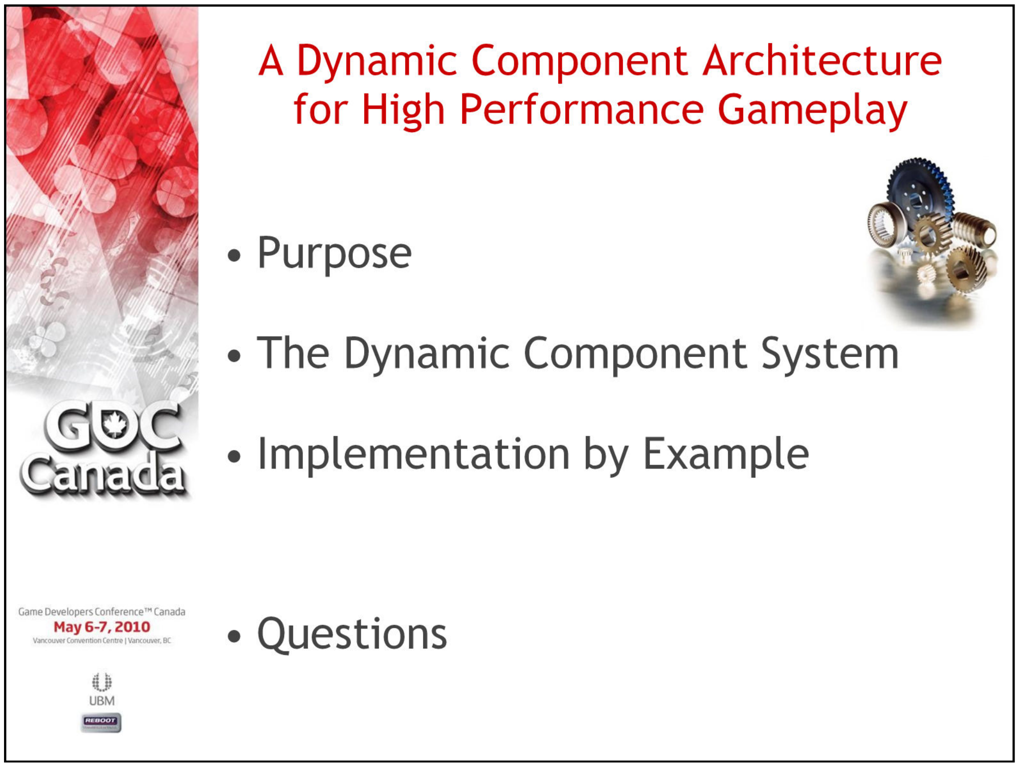
http://twitter.com/terrance_cohen

Connect with me on LinkedIn:

http://www.linkedin.com/in/terrancecohen

Ask me a question:

http://www.formspring.me/terrancecohen



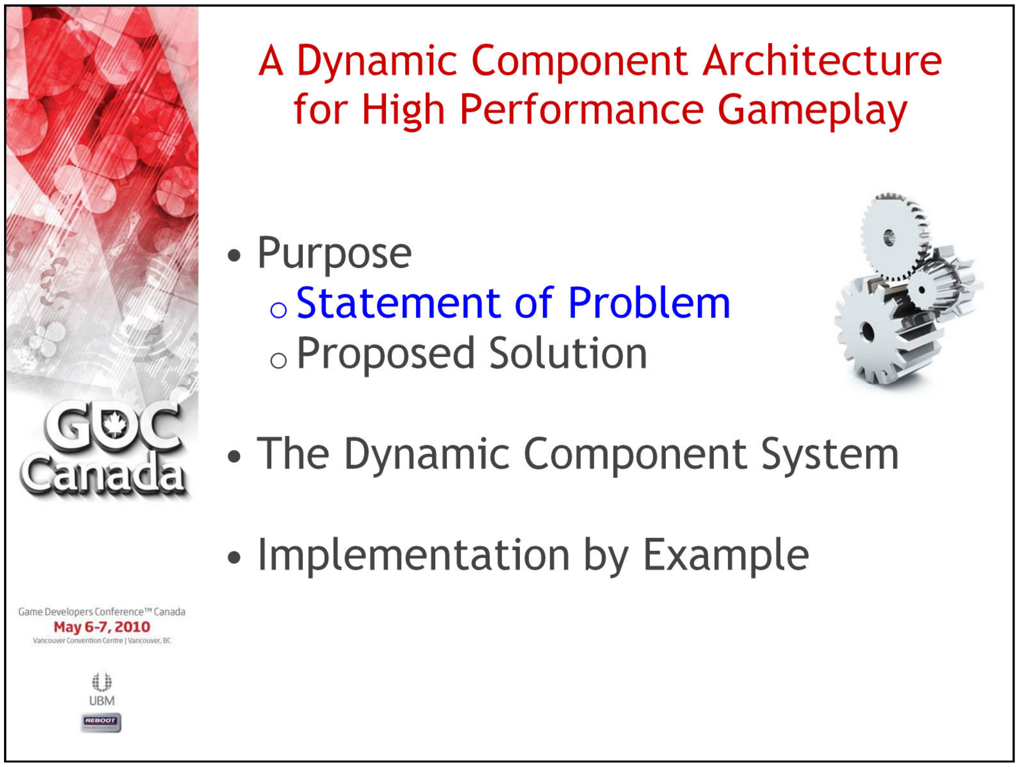
A Dynamic Component Architecture for High Performance Gameplay

- Purpose
- The Dynamic Component System
- Implementation by Example
- Questions

Game Developers Conference™ Canada
May 6-7, 2010
Vancouver Convention Centre | Vancouver, BC

- 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."
    - 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

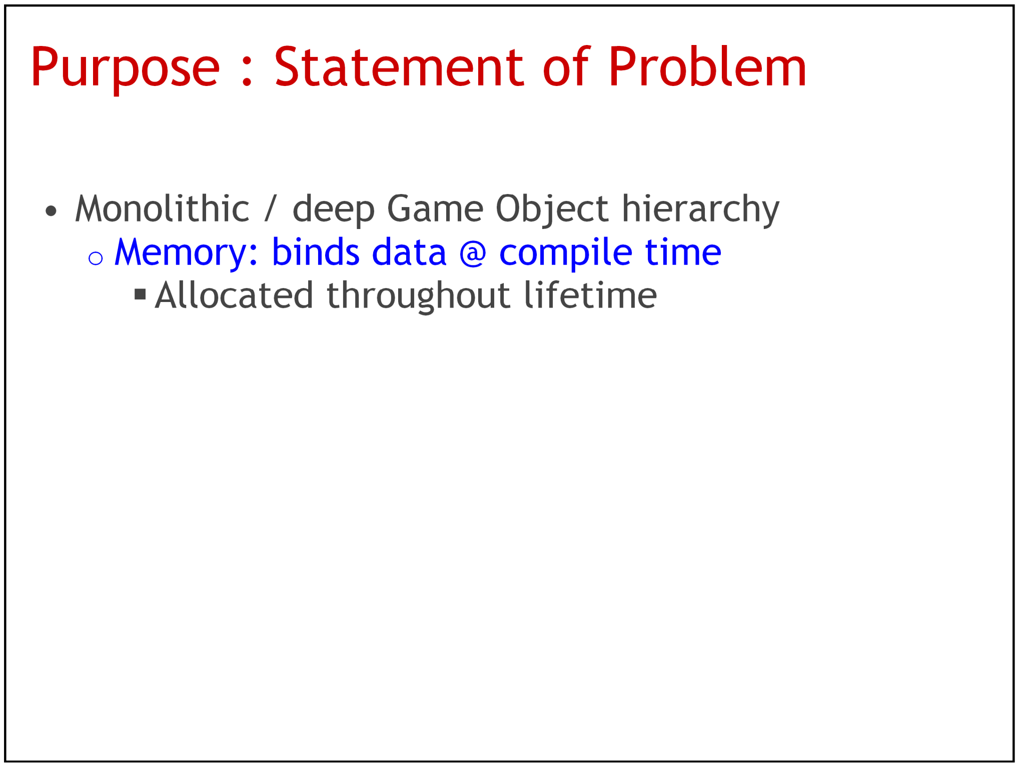


A Dynamic Component Architecture for High Performance Gameplay

- Purpose
  - Statement of Problem
  - Proposed Solution
- The Dynamic Component System
- Implementation by Example

Game Developers Conference™ Canada
May 6-7, 2010
Vancouver Convention Centre | Vancouver, BC

First we'll talk about the problem we're trying to solve.

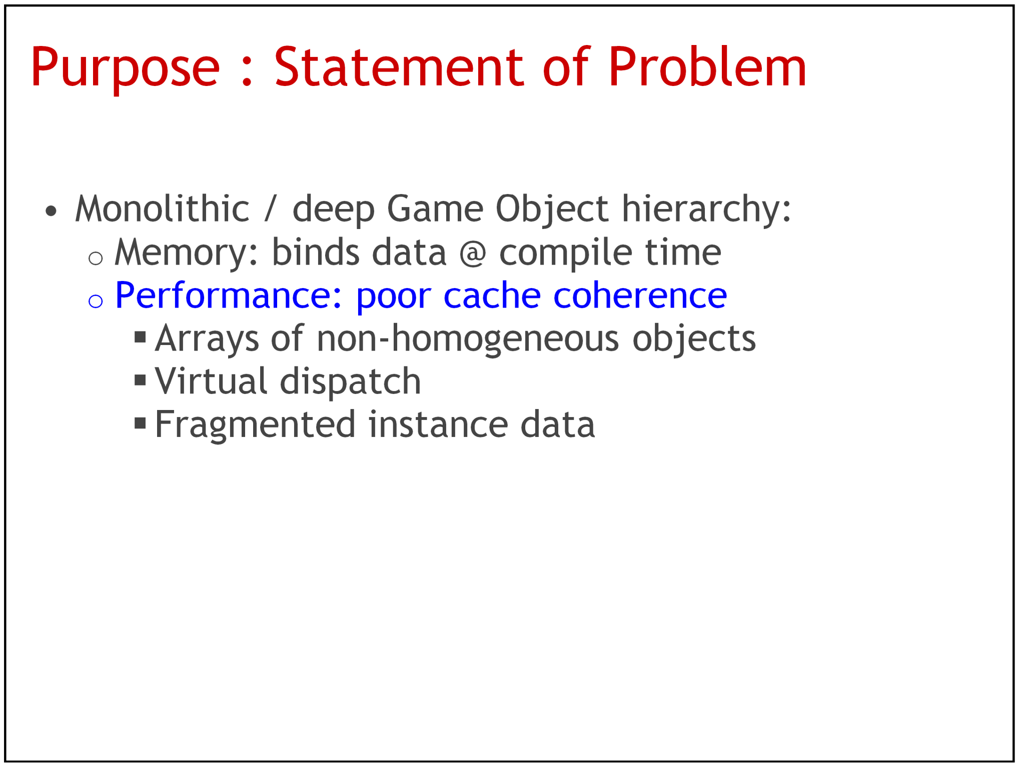


Purpose : Statement of Problem

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

- This is the traditional OO game object based gameplay model

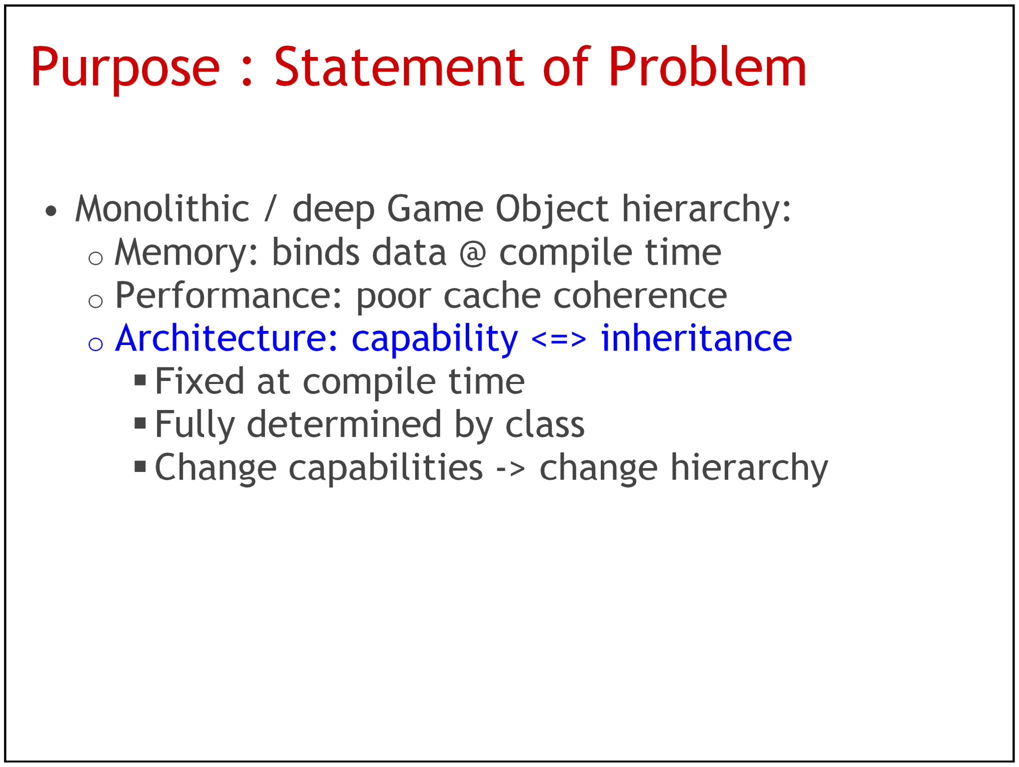
- Member data allocated throughout lifetime, even when not in use
  - Some designs will allocate some data separately from the game object, e.g. a physics instance
  - But this is counter to the architecture, rather than central to the architecture,
  - And each system is likely to roll it’s own solution, since the unifying architecture doesn’t support it



Purpose : Statement of Problem

- Monolithic / deep Game Object hierarchy:
  - Memory: binds data @ compile time
  - Performance: poor cache coherence
    - Arrays of non-homogeneous objects
    - Virtual dispatch
    - Fragmented instance data

- 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

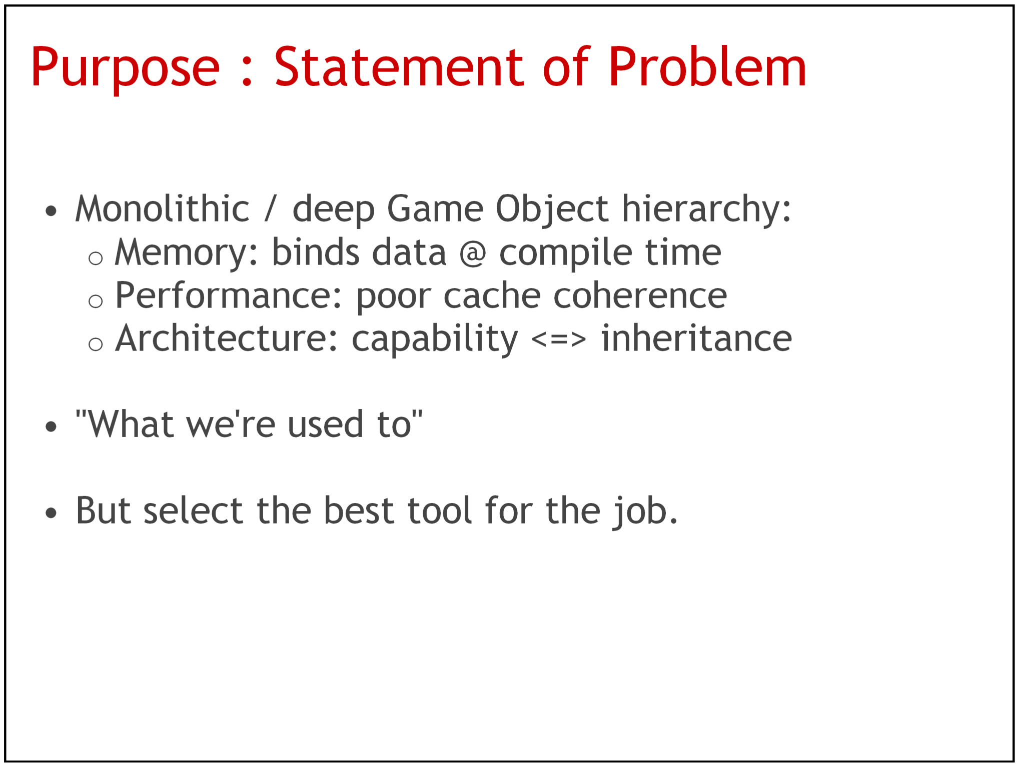


Purpose : Statement of Problem

- Monolithic / deep Game Object hierarchy:
  - Memory: binds data @ compile time
  - Performance: poor cache coherence
  - Architecture: capability <=> inheritance
    - Fixed at compile time
    - Fully determined by class
    - Change capabilities -> change hierarchy

- 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.

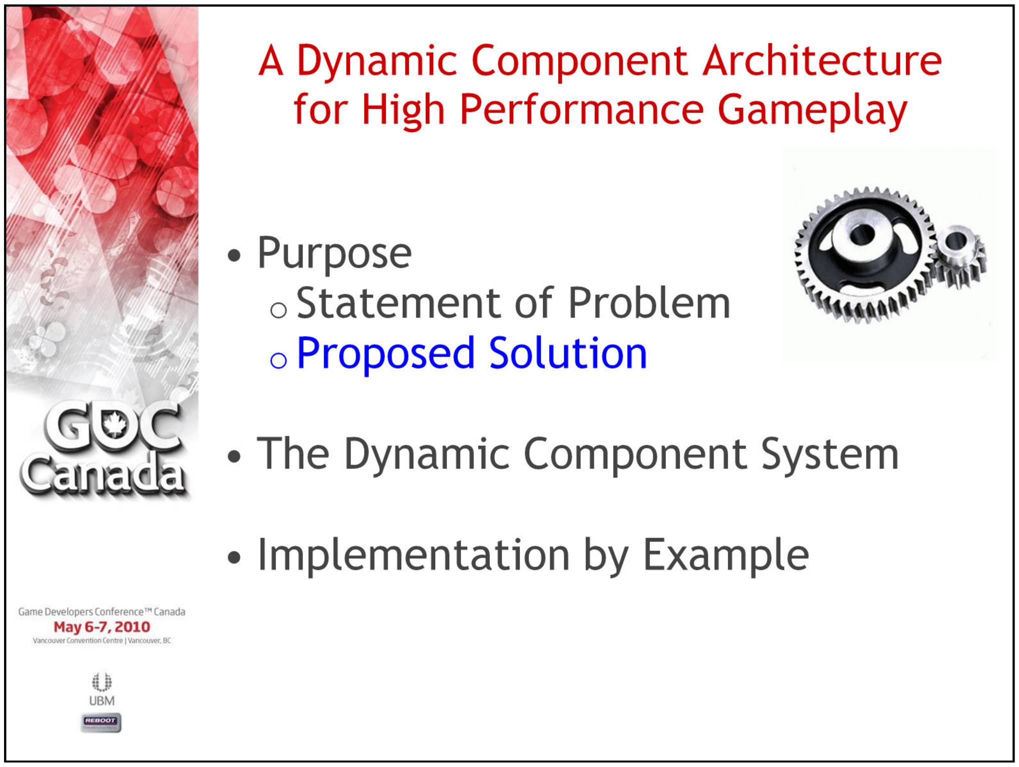
Purpose : Statement of Problem

- 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!



A Dynamic Component Architecture for High Performance Gameplay

- Purpose
  - Statement of Problem
  - Proposed Solution
- The Dynamic Component System
- Implementation by Example

Game Developers Conference™ Canada
May 6-7, 2010
Vancouver Convention Centre | Vancouver, BC

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

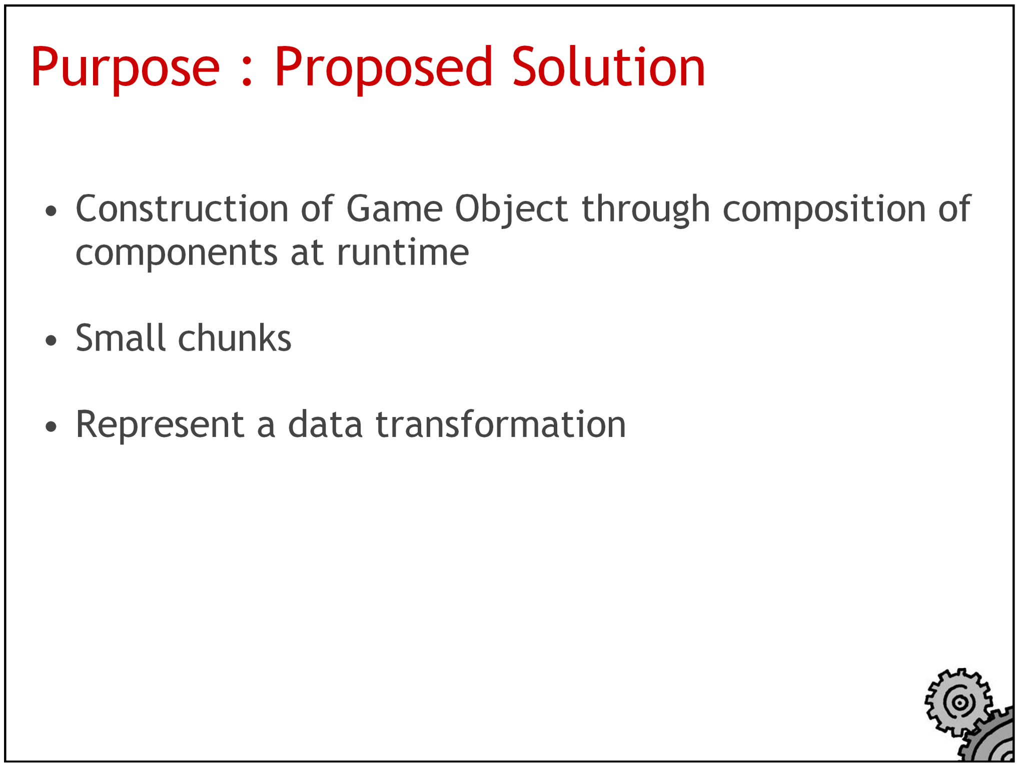
- Construction of Game Object through composition of components at runtime

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 !?!

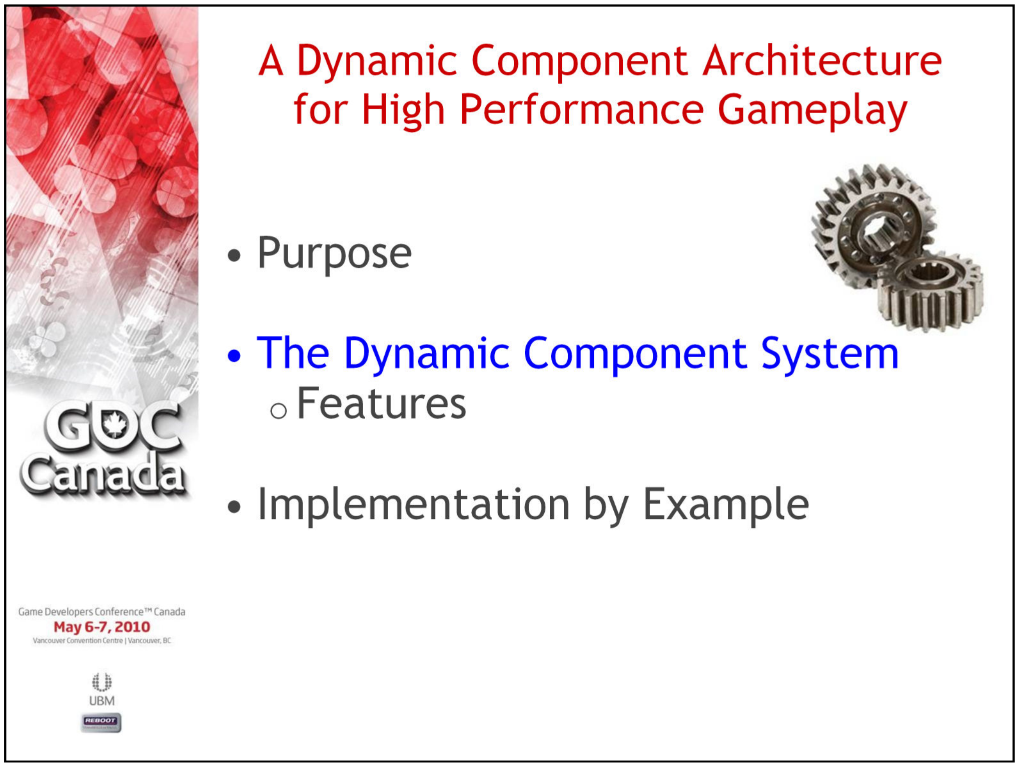


Purpose : Proposed Solution

- Construction of Game Object through composition of components at runtime
- Small chunks
- Represent a data transformation

- 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
  - e.g. state machine + states

e.g.
- model
- animation
- physics
- ai/logic
- sensing
- send/receive messaging
- causing damage



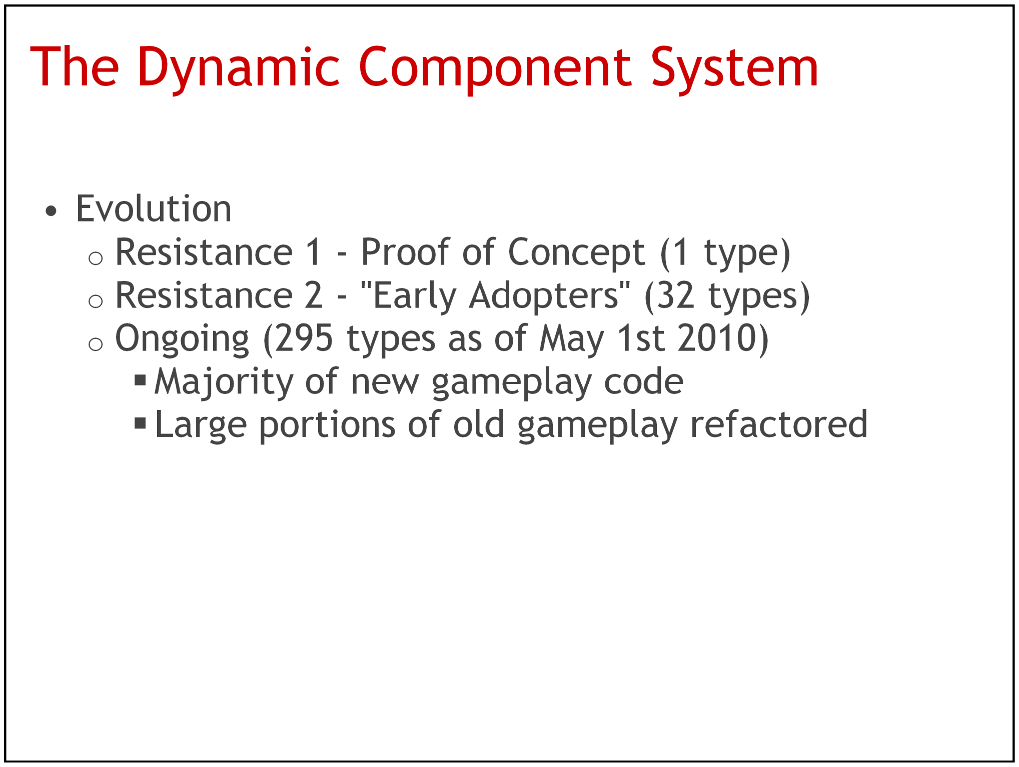
A Dynamic Component Architecture for High Performance Gameplay

- Purpose
- The Dynamic Component System
  - Features
- Implementation by Example

GDC Canada
Game Developers Conference™ Canada
May 6-7, 2010
Vancouver Convention Centre | Vancouver, BC
UBM
REBOOT

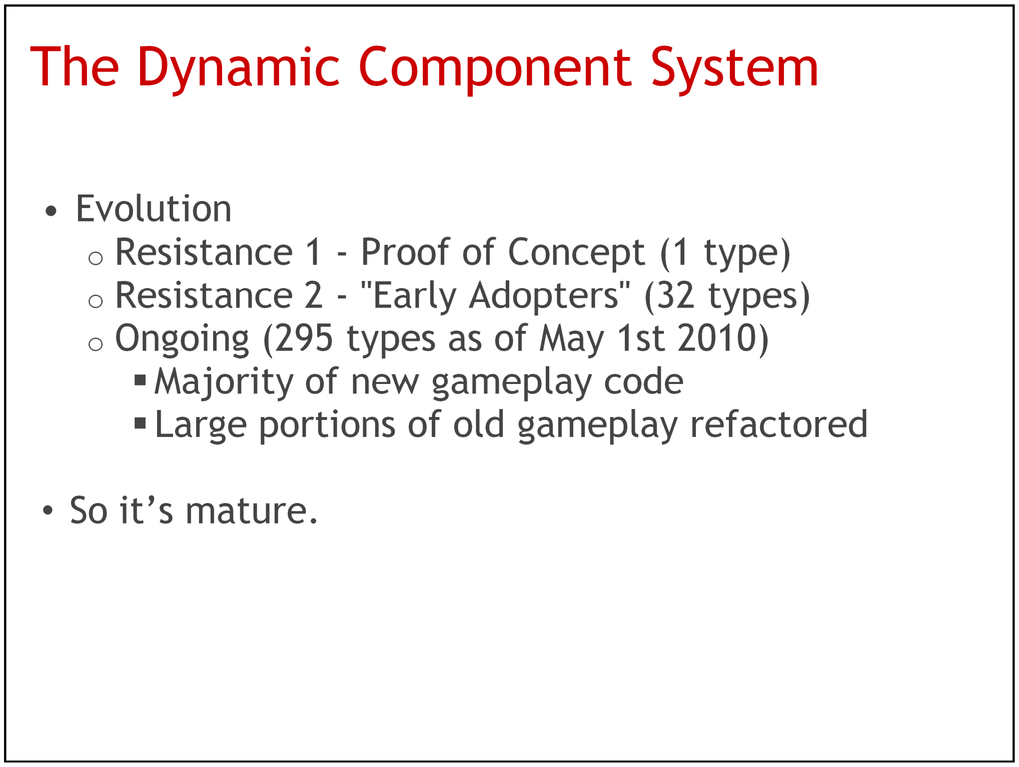
So now we'll talk about my solution which I call the Dynamic Component System.

First we'll talk at a high level about what is the system



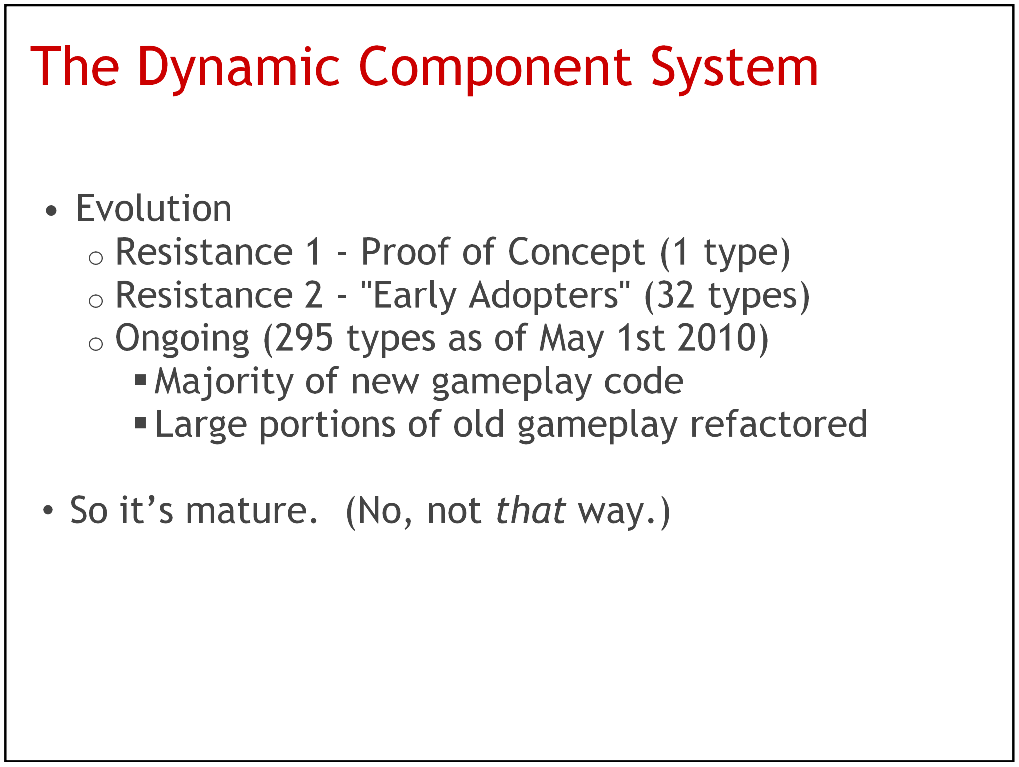
The Dynamic Component System

- 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

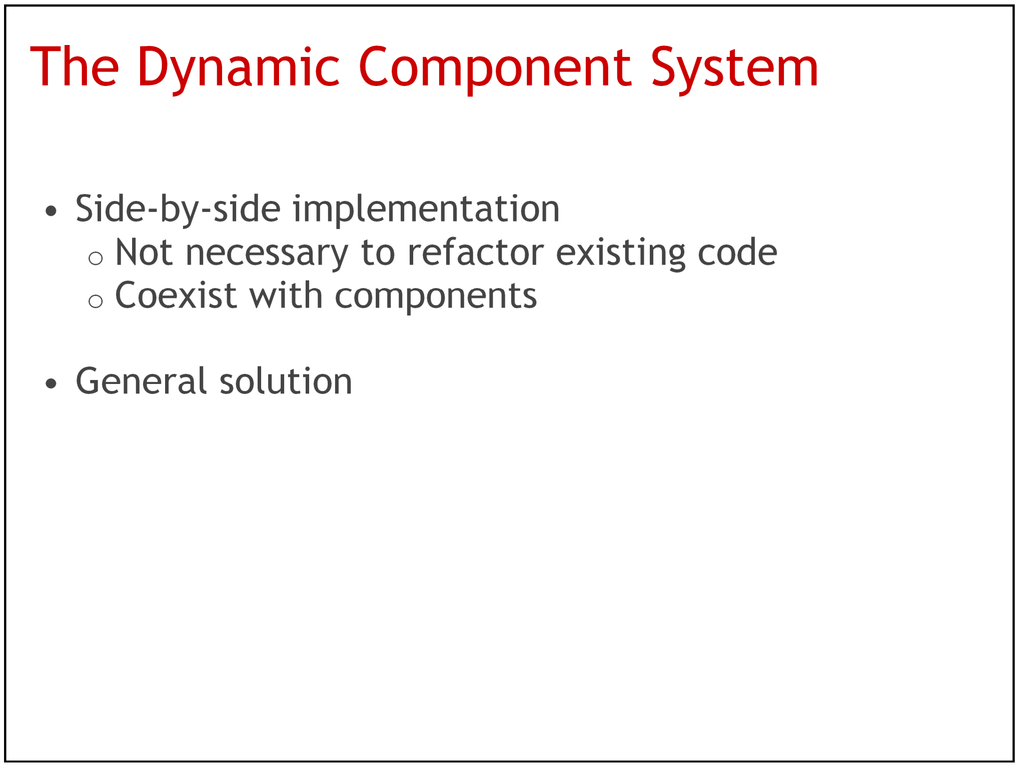
- Evolution
  - Resistance 1 - Proof of Concept (1 type)
  - Resistance 2 - "Early Adopters" (32 types)
  - Ongoing (295 types as of May 1st 2010)
    - Majority of new gameplay code
    - Large portions of old gameplay refactored
- So it’s mature.



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.



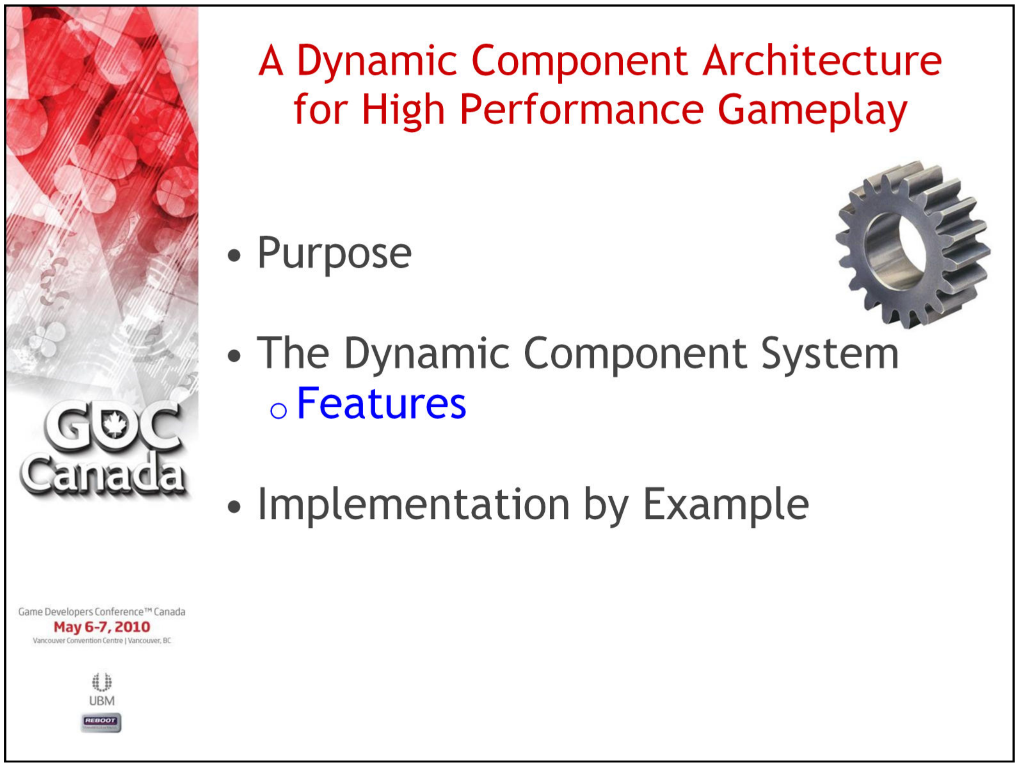
The Dynamic Component System

- 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
  - Reflection
  - Serialization
  - Data building
  - Instance versioning
  - ... those things are handled separately
    - outside the scope of this discussion



A Dynamic Component Architecture for High Performance Gameplay

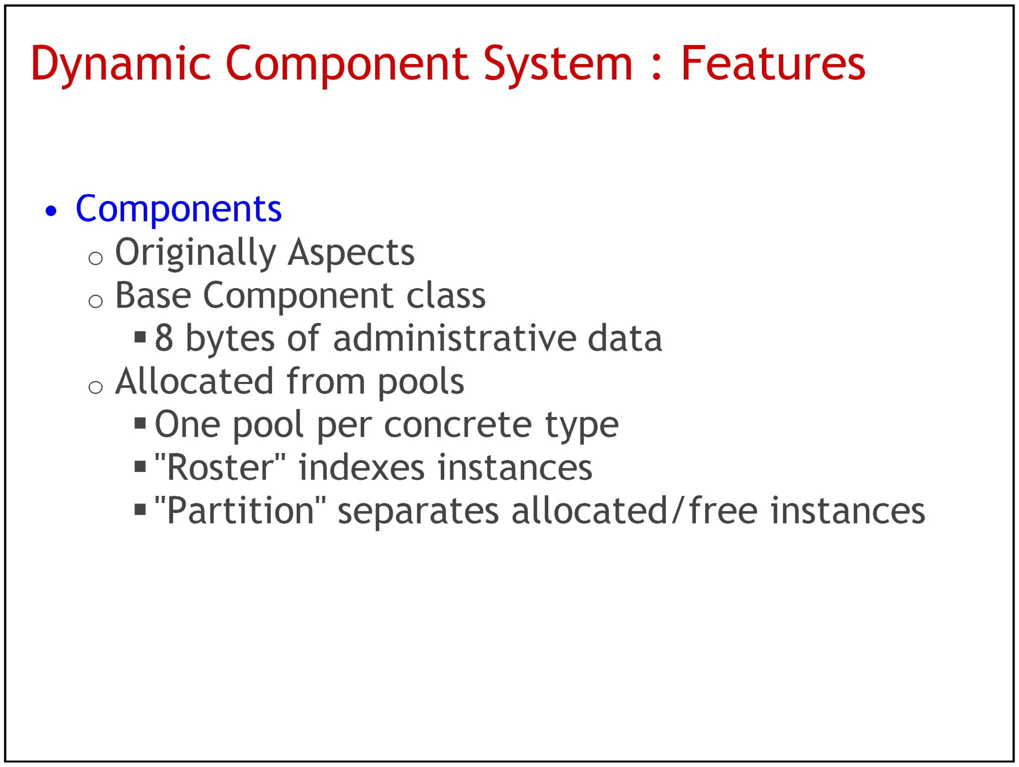
- Purpose
- The Dynamic Component System
  - Features
- Implementation by Example

Game Developers Conference™ Canada
May 6-7, 2010
Vancouver Convention Centre | Vancouver, BC

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

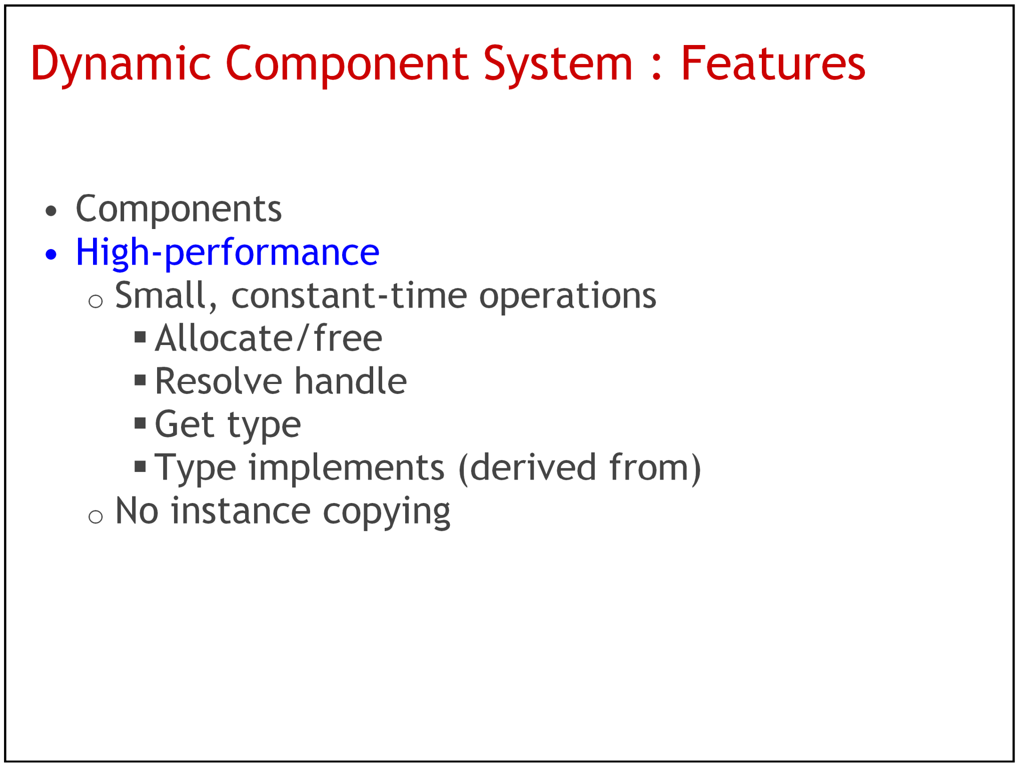


Dynamic Component System : Features

- Components
  - Originally Aspects
  - Base Component class
    - 8 bytes of administrative data
  - Allocated from pools
    - One pool per concrete type
    - "Roster" indexes instances
    - "Partition" separates allocated/free instances

Components

- Originally called Aspects from AOP
- Derived from base Component class
  - 12 bytes of administrative data
  - "Payload" can be detached & treated as POD for processing in separate memory spaces
- Allocated from pools
  - One pool per concrete type
    - Homogeneous type instances per pool
- Each pool is indexed with a "partition" value separating free from allocated instances
  - Called the "Roster"

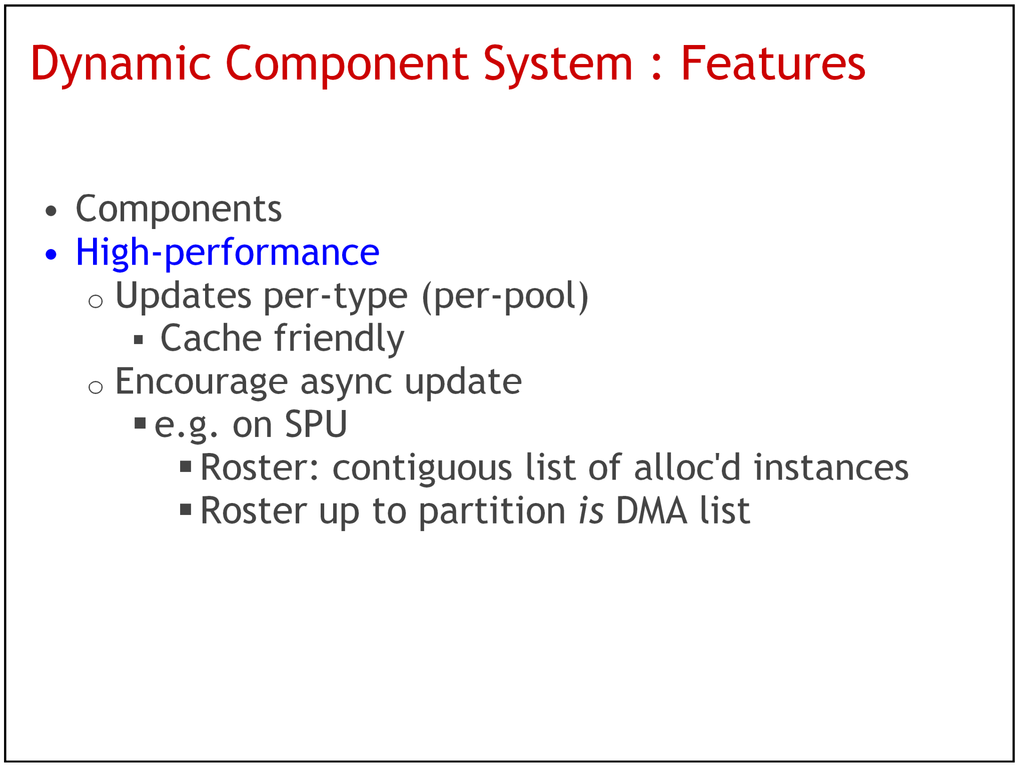


Dynamic Component System : Features

- Components
- High-performance
  - Small, constant-time operations
    - Allocate/free
    - Resolve handle
    - Get type
    - Type implements (derived from)
  - No instance copying

High-performance

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

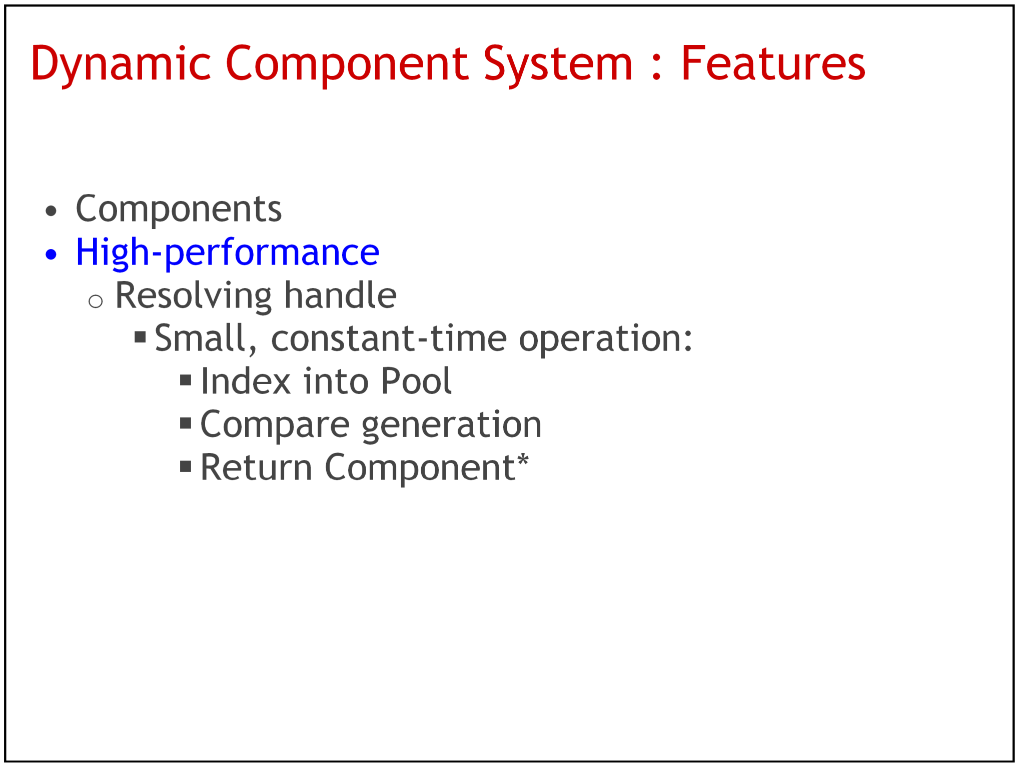


Dynamic Component System : Features

- 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

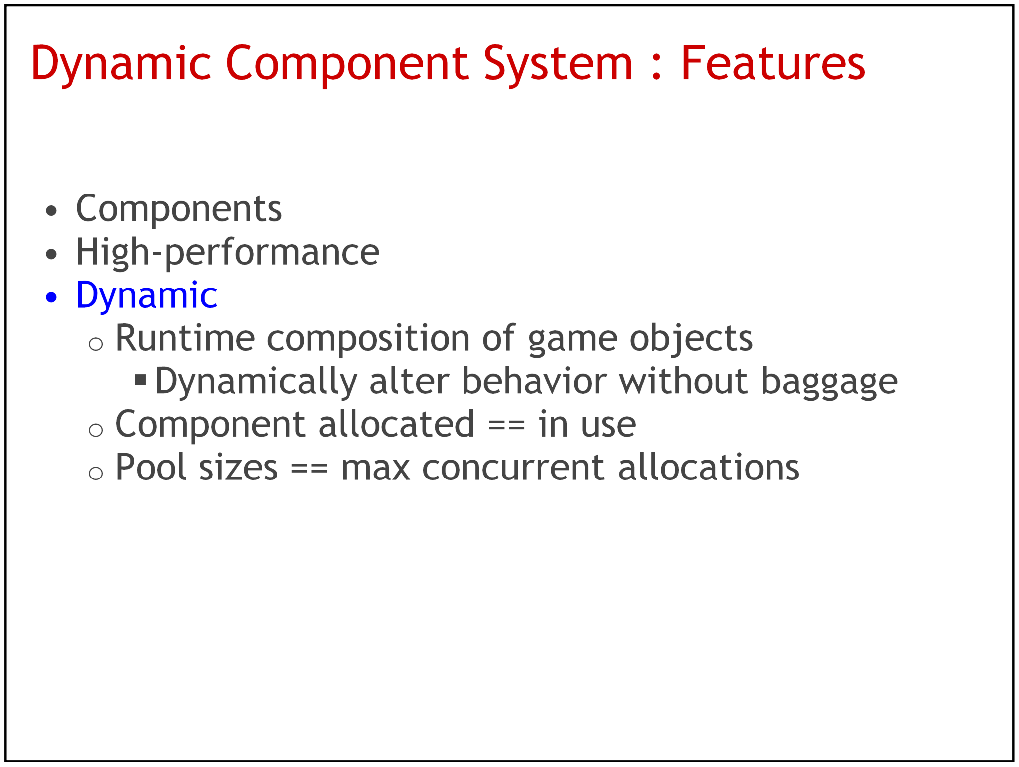


Dynamic Component System : Features

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

High-performance:

- References between components normally handled by holding/resolving handles

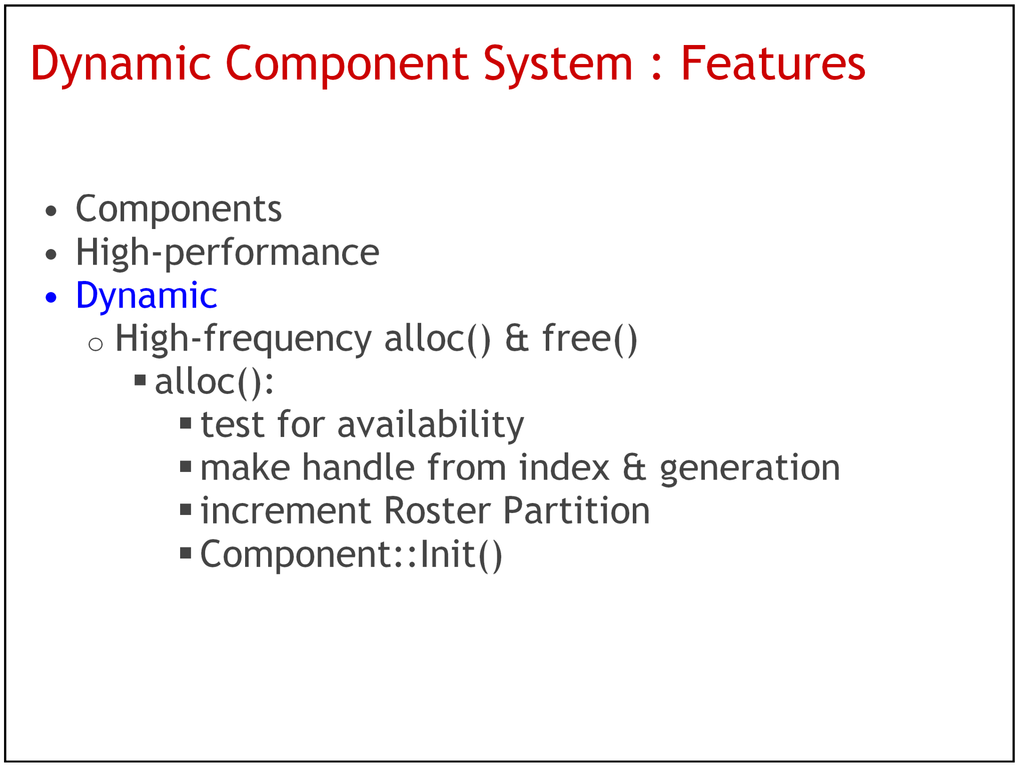


Dynamic Component System : Features

- 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

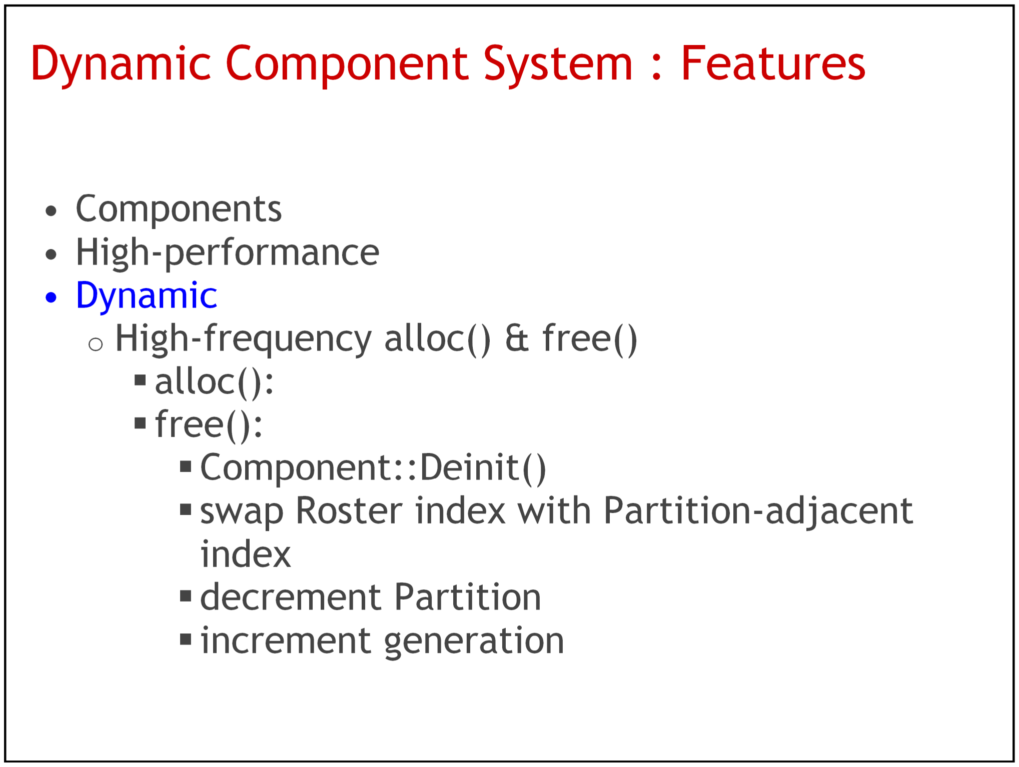


Dynamic Component System : Features

- 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

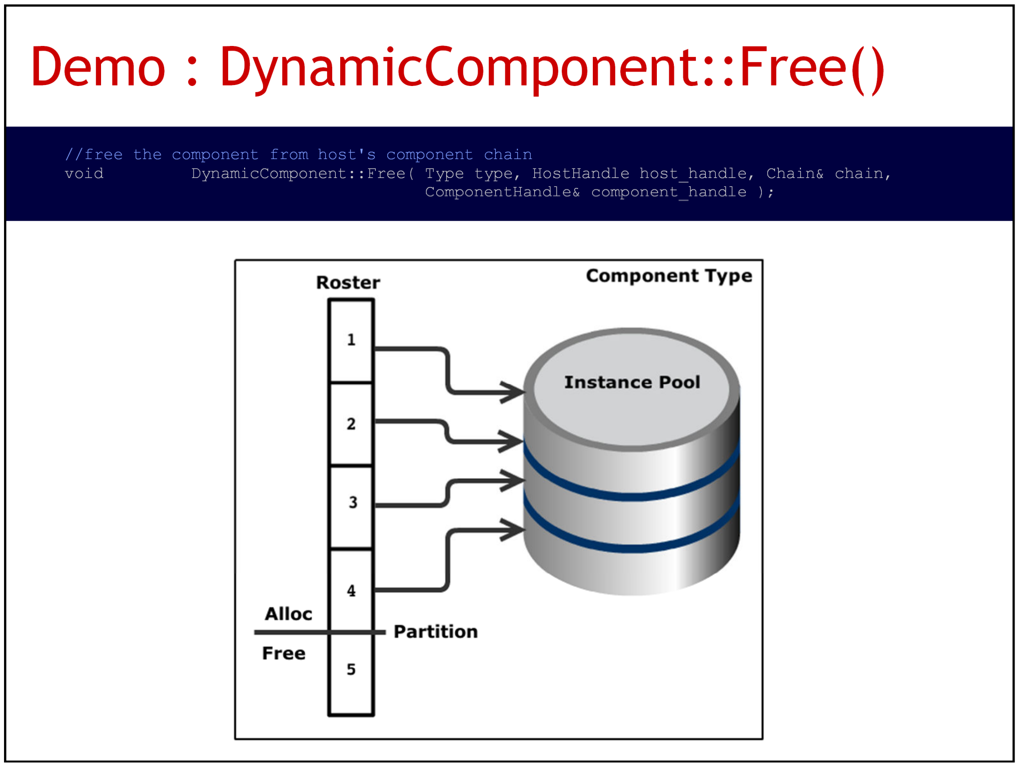
- Components
- High-performance
- Dynamic
  - High-frequency alloc() & free()
    - alloc():
    - free():
      - Component::Deinit()
      - swap Roster index with Partition-adjacent index
      - decrement Partition
      - increment generation

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

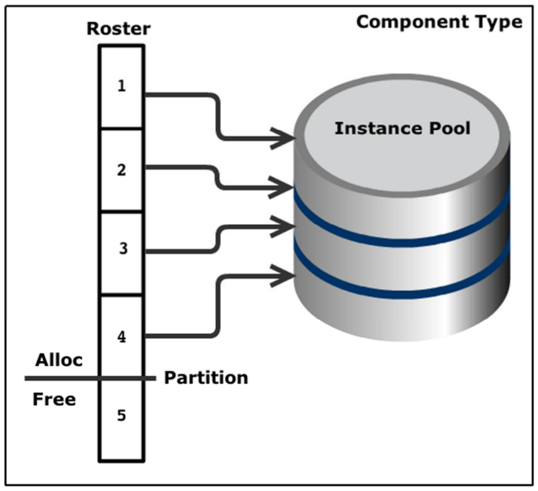
So let's take an example to put some of these pieces together.

We'll look at an example of freeing a dynamic component instance.



Demo : DynamicComponent::Free()

```
//free the component from host's component chain
void         DynamicComponent::Free( Type type, HostHandle host_handle, Chain& chain,
                                     ComponentHandle& component_handle );
```



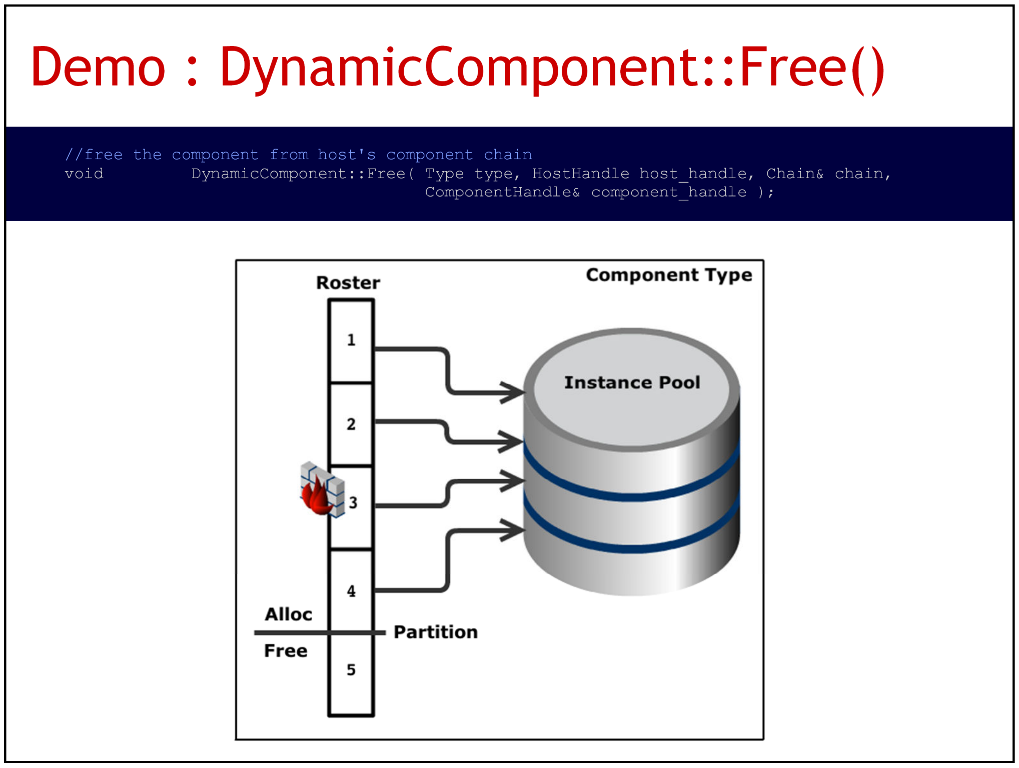
Here's the prototype for DynamicComponent::Free(). It takes...

So here's an example.

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

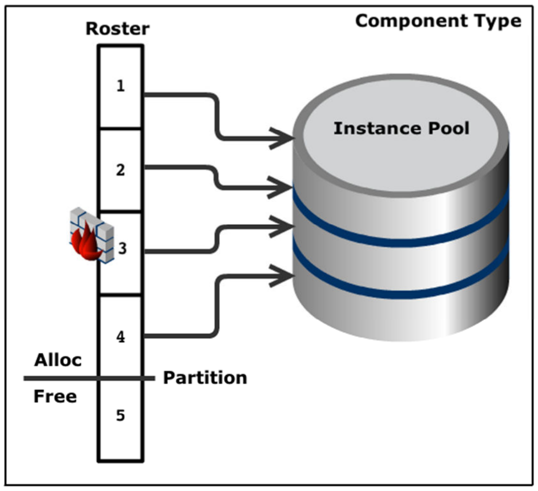
We have a roster, which is an array of indices into the instance pool. (Note that all of the instances in the pool are of the same type, so they are of equal size. So the value at a roster index is itself just an index into the pool.

You can see that we have a partition value, which separates roster indices representing allocated vs. free instances.

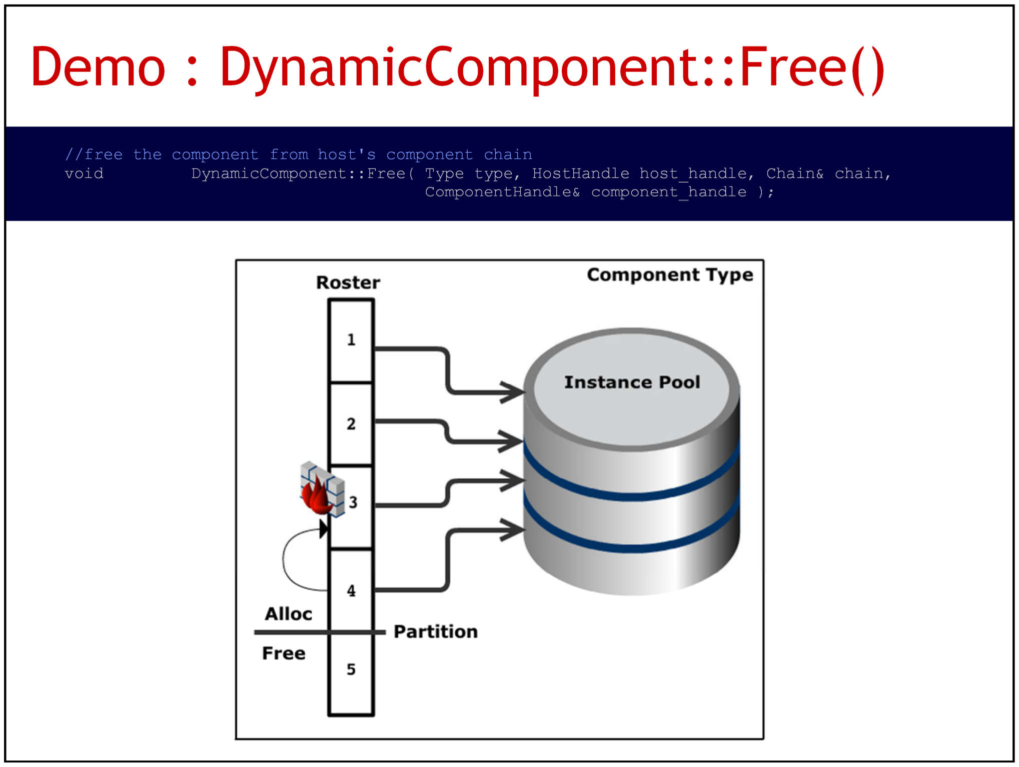


Demo : DynamicComponent::Free()

```
//free the component from host's component chain
void         DynamicComponent::Free( Type type, HostHandle host_handle, Chain& chain,
                                     ComponentHandle& component_handle );
```

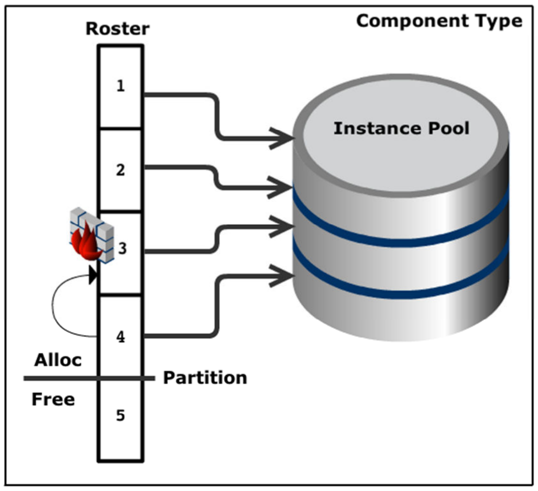


Alright. Now we're going to free the component represented by the 3rd roster element, which is currently index 3 in the pool.



Demo : DynamicComponent::Free()

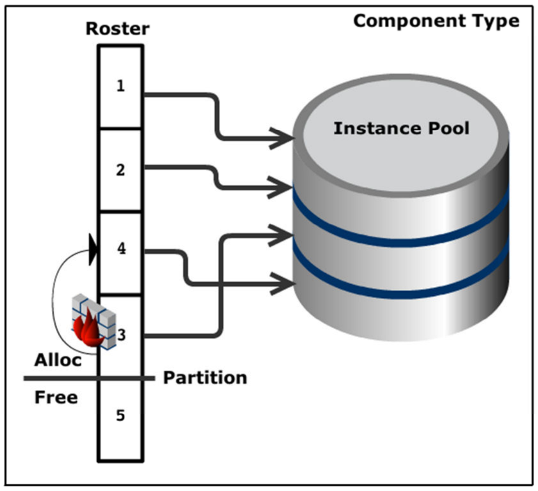
```
//free the component from host's component chain
void         DynamicComponent::Free( Type type, HostHandle host_handle, Chain& chain,
                                     ComponentHandle& component_handle );
```



So what we'll do is swap the 3rd and 4th roster elements.

Demo : DynamicComponent::Free()

```
//free the component from host's component chain
void         DynamicComponent::Free( Type type, HostHandle host_handle, Chain& chain,
                                     ComponentHandle& component_handle );
```

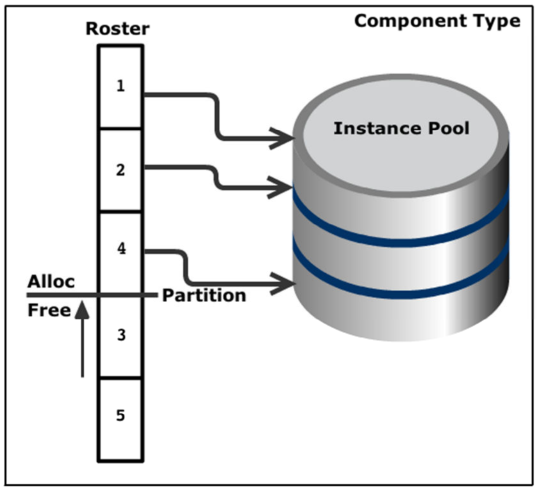


Now we're freeing the 4th roster element, which represents the 3rd instance in the pool.

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...

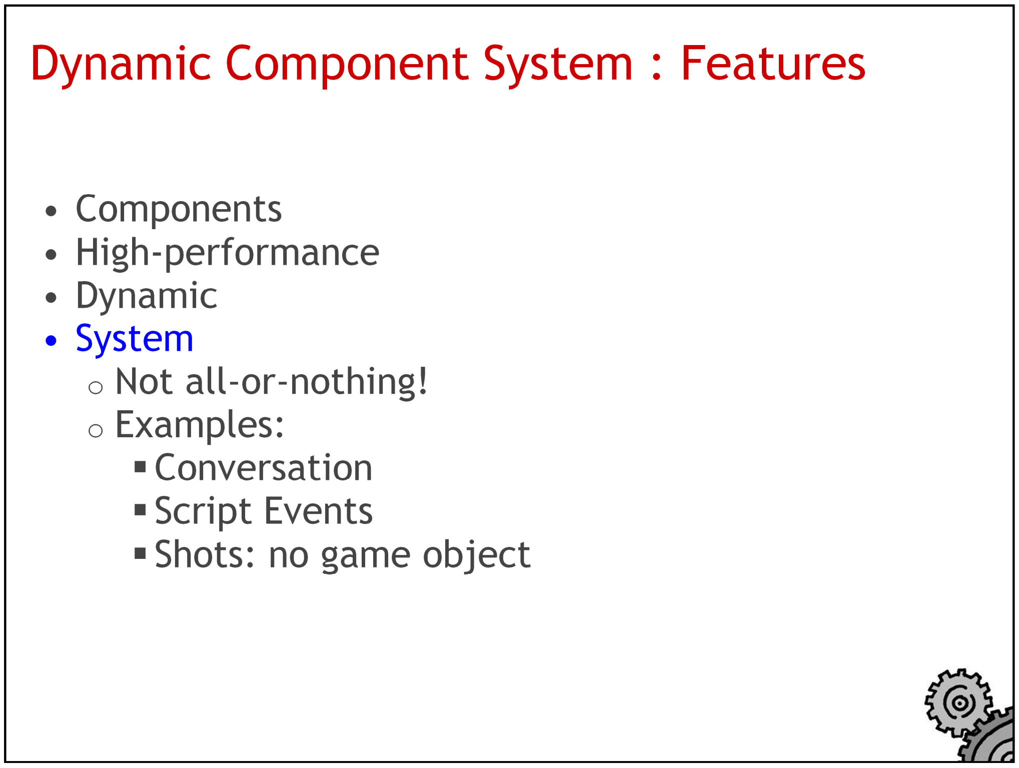
Demo : DynamicComponent::Free()

```
//free the component from host's component chain
void        DynamicComponent::Free( Type type, HostHandle host_handle, Chain& chain,
                                    ComponentHandle& component_handle );
```



... 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.

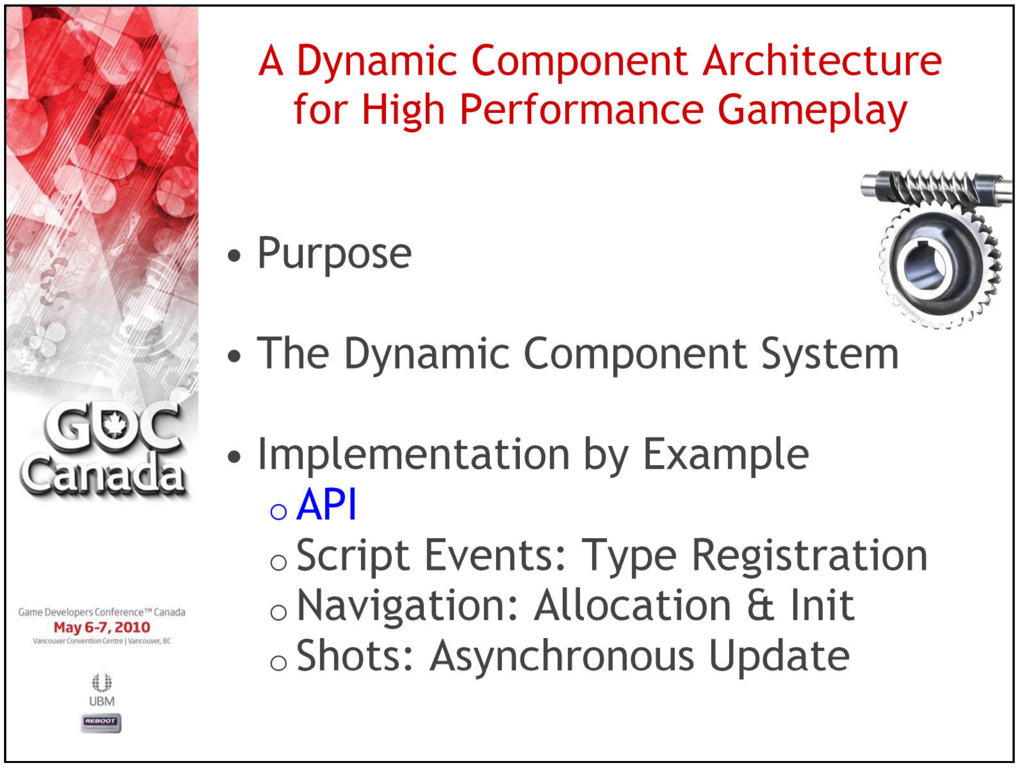


Dynamic Component System : Features

- 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

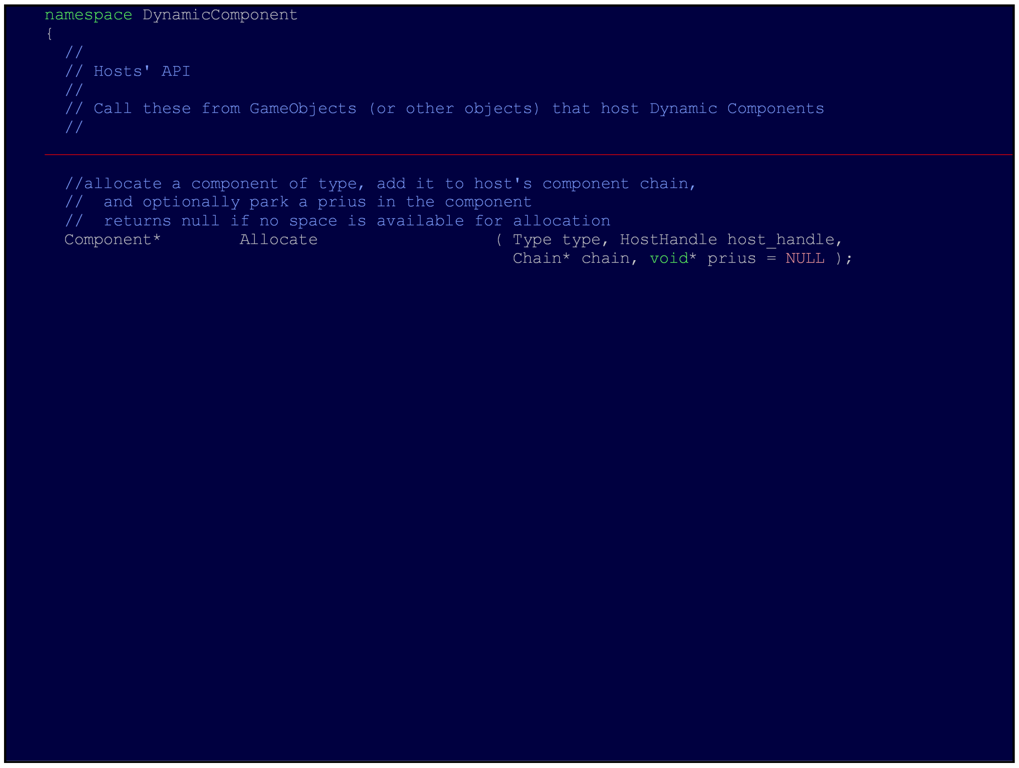

A Dynamic Component Architecture for High Performance Gameplay

- Purpose
- The Dynamic Component System
- Implementation by Example
  - API
  - Script Events: Type Registration
  - Navigation: Allocation & Init
  - Shots: Asynchronous Update

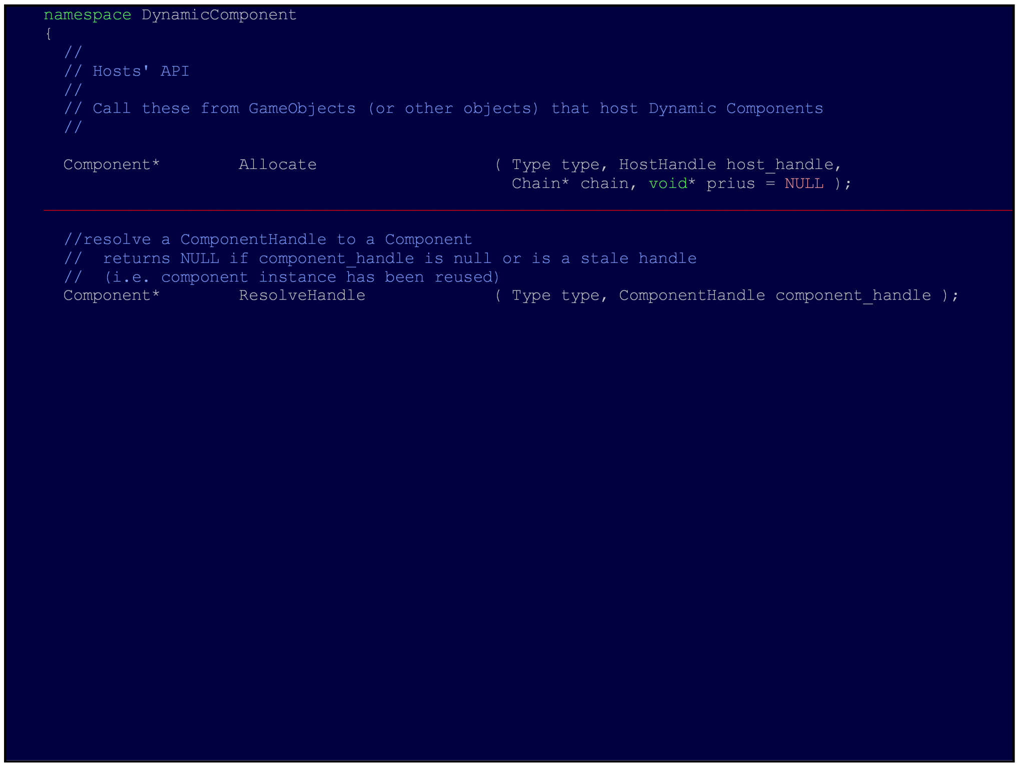
GDC Canada
Game Developers Conference™ Canada
May 6-7, 2010
Vancouver Convention Centre | Vancouver, BC
UBM
REBOOT

- At a high level, I have three sections to this discussion

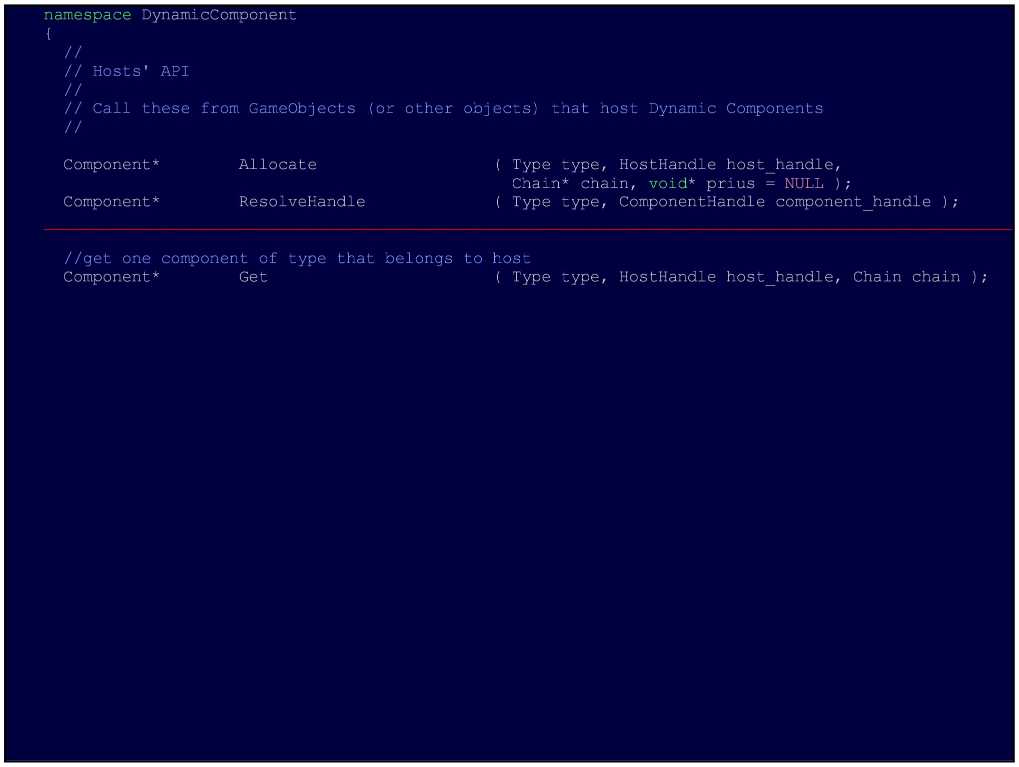
1. So now we're digging into implementation details of the system
   - APIs always a good place to start



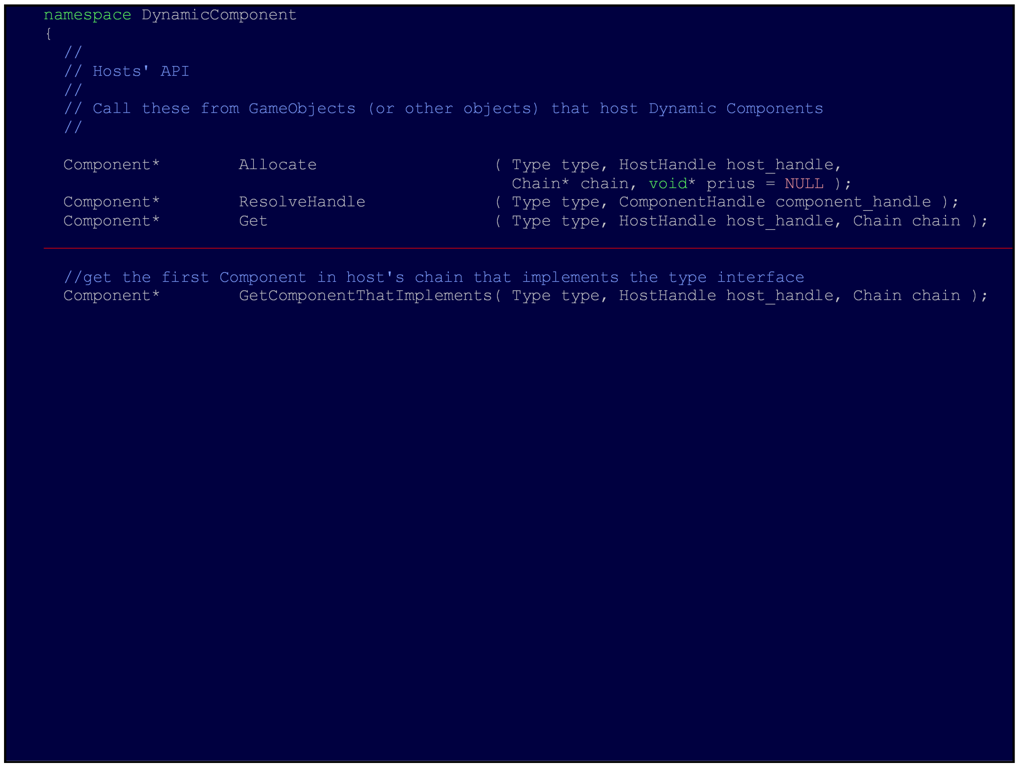
```
namespace DynamicComponent
{
  //
  // Hosts' API
  //
  // Call these from GameObjects (or other objects) that host Dynamic Components
  //

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



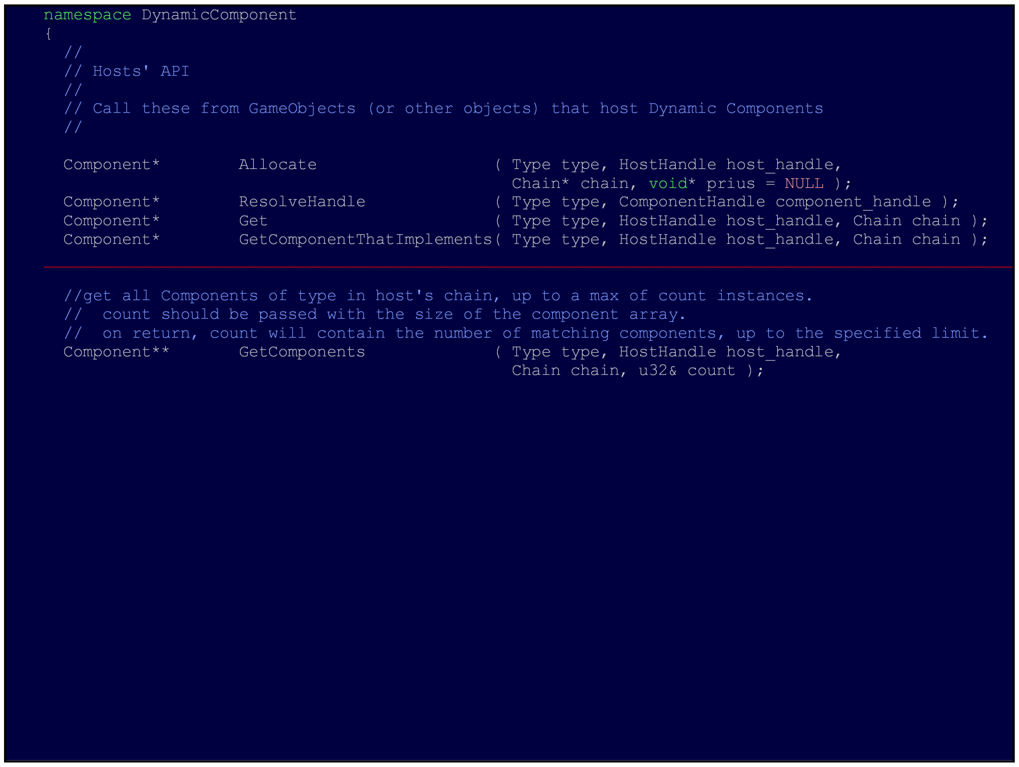
```
namespace DynamicComponent
{
  //
  // Hosts' API
  //
  // Call these from GameObjects (or other objects) that host Dynamic Components
  //

  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 );
```



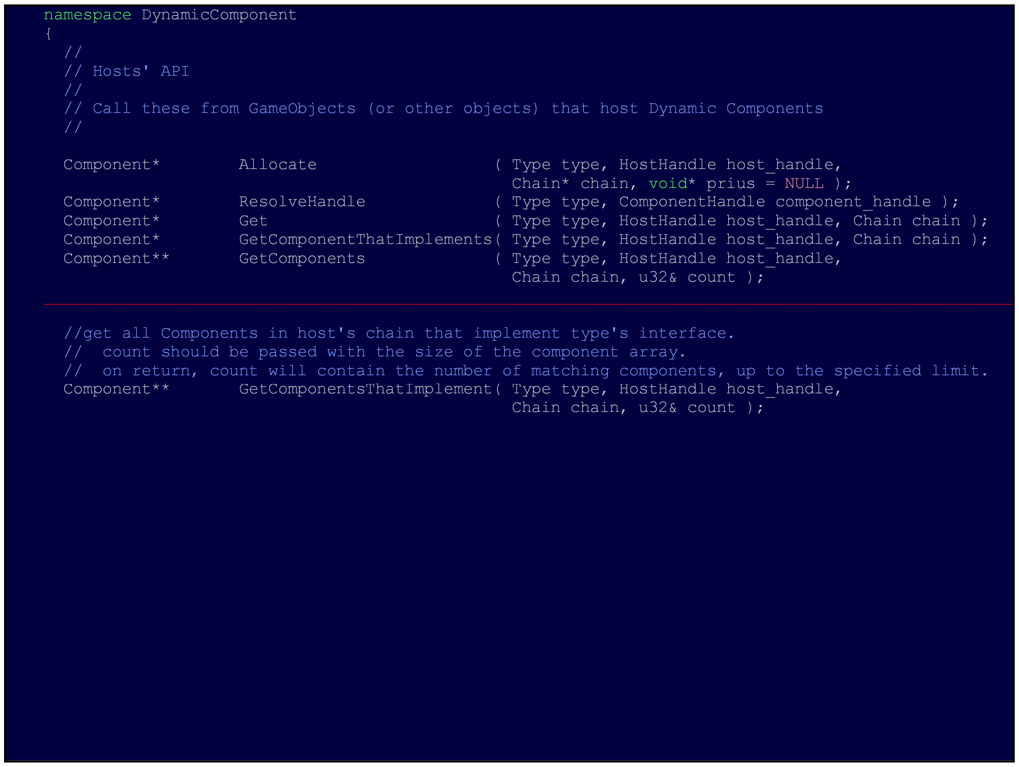
```
namespace DynamicComponent
{
  //
  // Hosts' API
  //
  // Call these from GameObjects (or other objects) that host Dynamic Components
  //

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



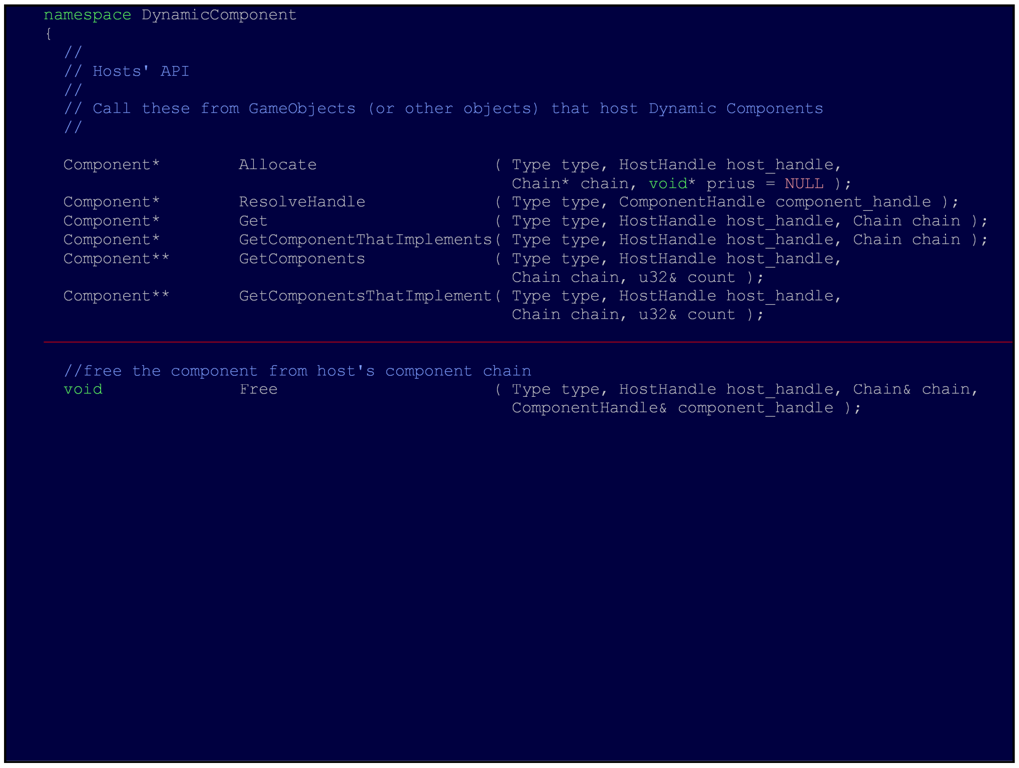
```
namespace DynamicComponent
{
  //
  // Hosts' API
  //
  // Call these from GameObjects (or other objects) that host Dynamic Components
  //

  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 );

  //get the first Component in host's chain that implements the type interface
  Component*        GetComponentThatImplements( Type type, HostHandle host_handle, Chain chain );
```



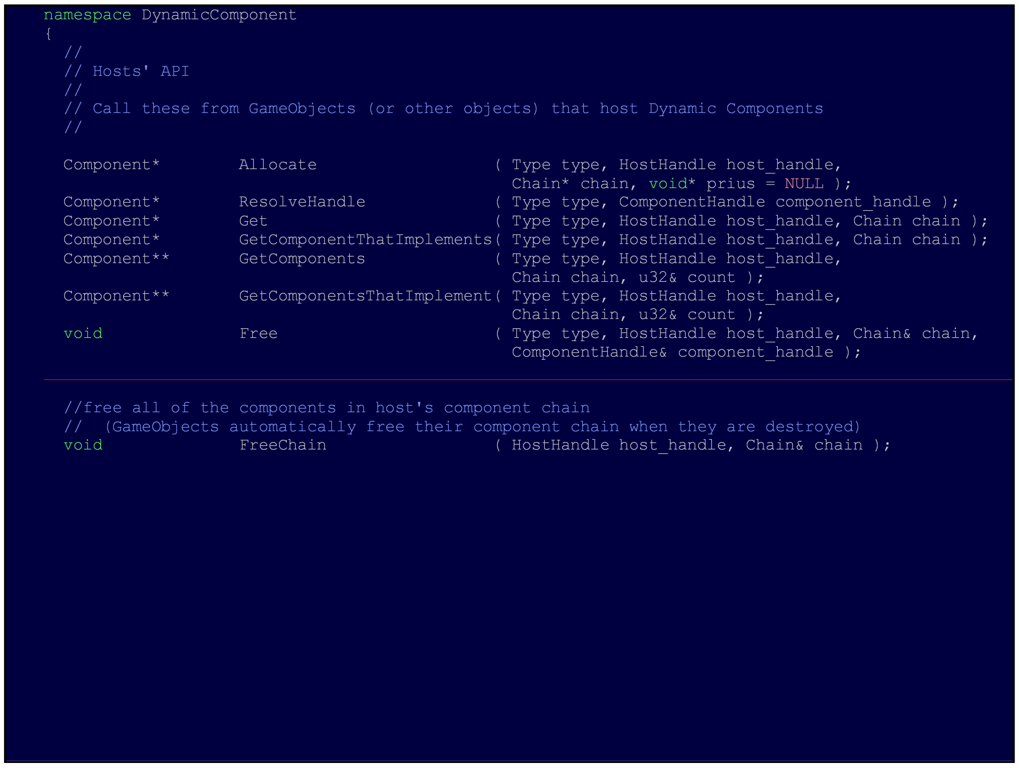
```
namespace DynamicComponent
{
  //
  // Hosts' API
  //
  // Call these from GameObjects (or other objects) that host Dynamic Components
  //

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

  //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, 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 );

  //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 );
```

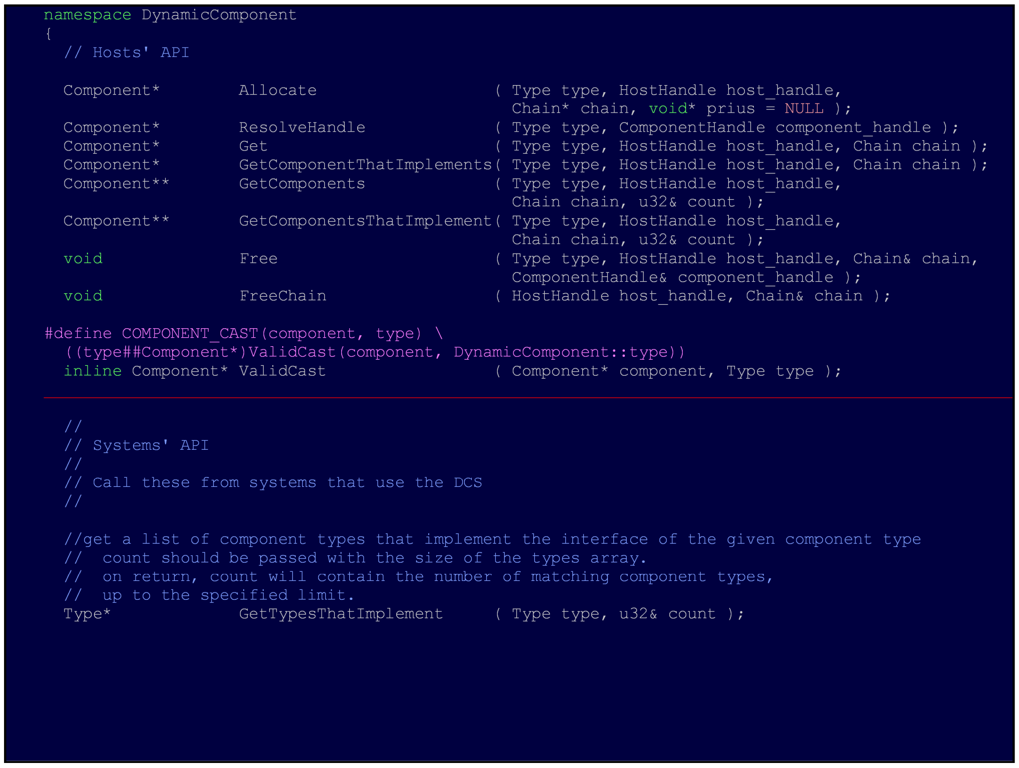


```
namespace DynamicComponent
{
  //
  // Hosts' API
  //
  // Call these from GameObjects (or other objects) that host Dynamic Components
  //

  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**       GetComponentsThatImplement( 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 );
```

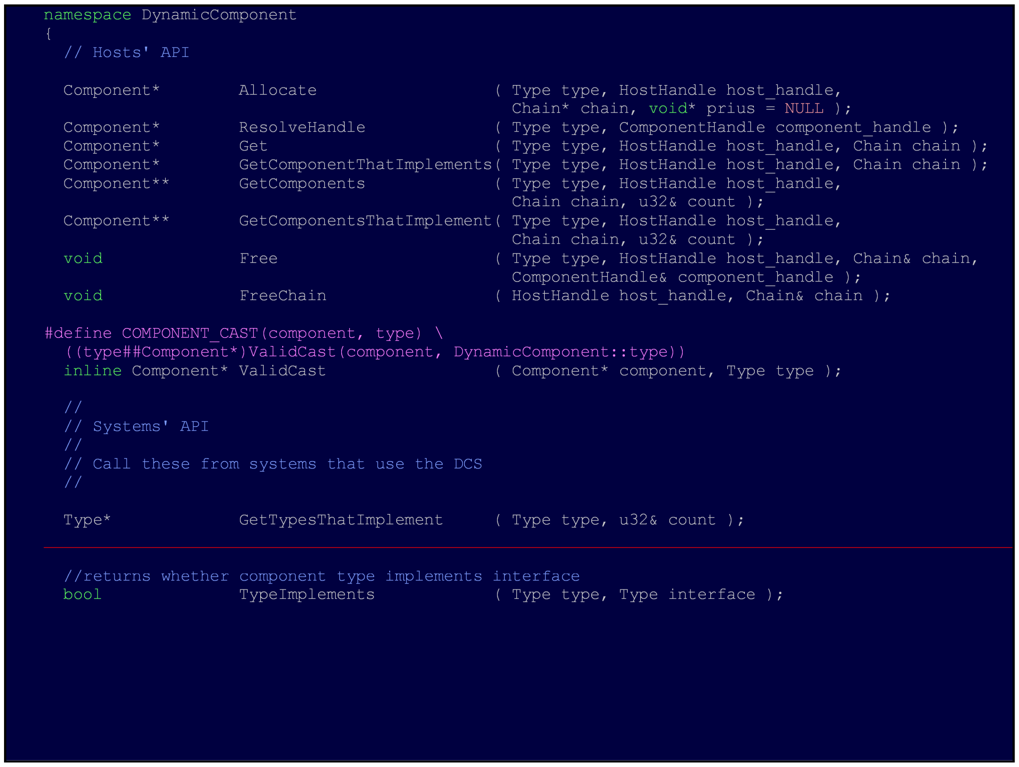


```
namespace DynamicComponent
{
  //
  // Hosts' API
  //
  // Call these from GameObjects (or other objects) that host Dynamic Components
  //

  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**       GetComponentsThatImplement( 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
  //  (GameObjects automatically free their component chain when they are destroyed)
  void              FreeChain                 ( HostHandle host_handle, Chain& chain );
```

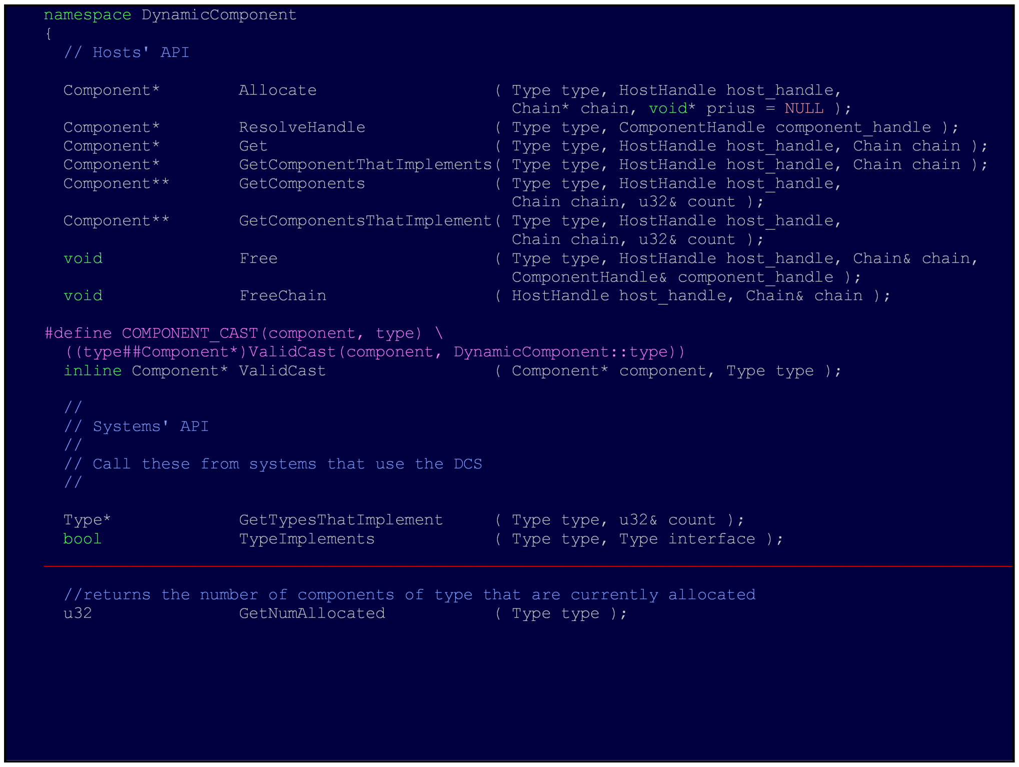
```
namespace DynamicComponent
{
  //
  // Hosts' API
  //
  // Call these from GameObjects (or other objects) that host Dynamic Components
  //

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

  //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:
  //  HeadComponent* my_head = COMPONENT_CAST(component_ptr, Head);
  //
#define COMPONENT_CAST(component, type) \
  ((type##Component*)ValidCast(component, DynamicComponent::type))
  inline Component* ValidCast                 ( Component* component, Type type );
```



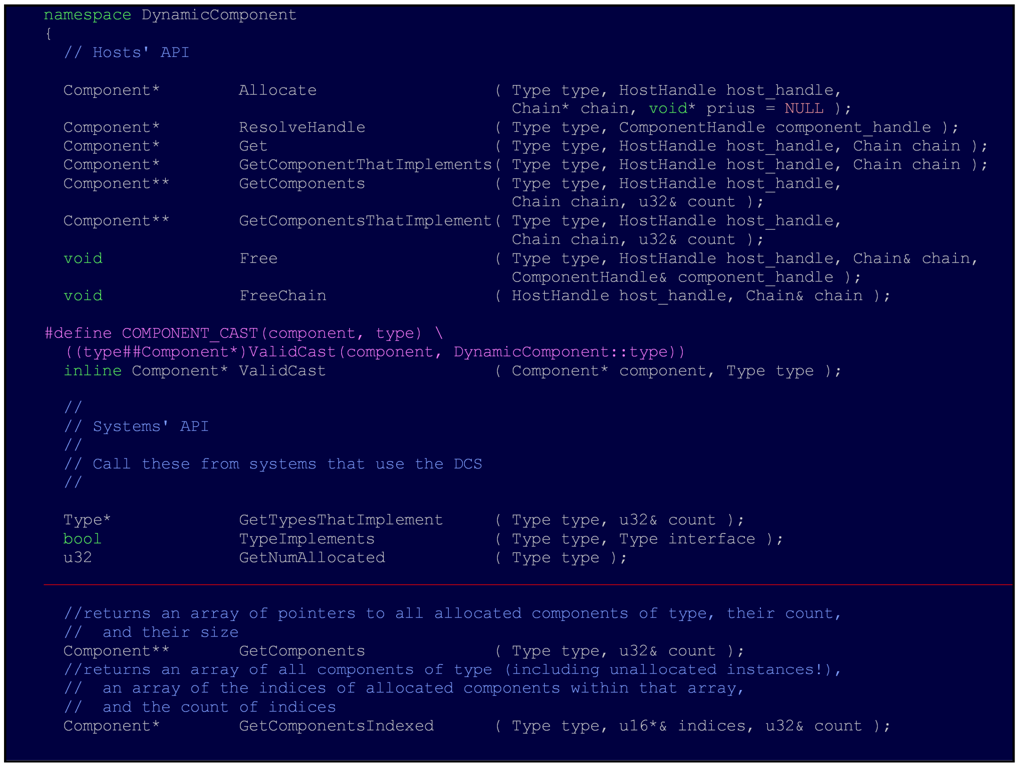
```
namespace DynamicComponent
{
  // Hosts' API

  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, 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
  //

  //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*             GetTypesThatImplement     ( Type type, u32& count );
```



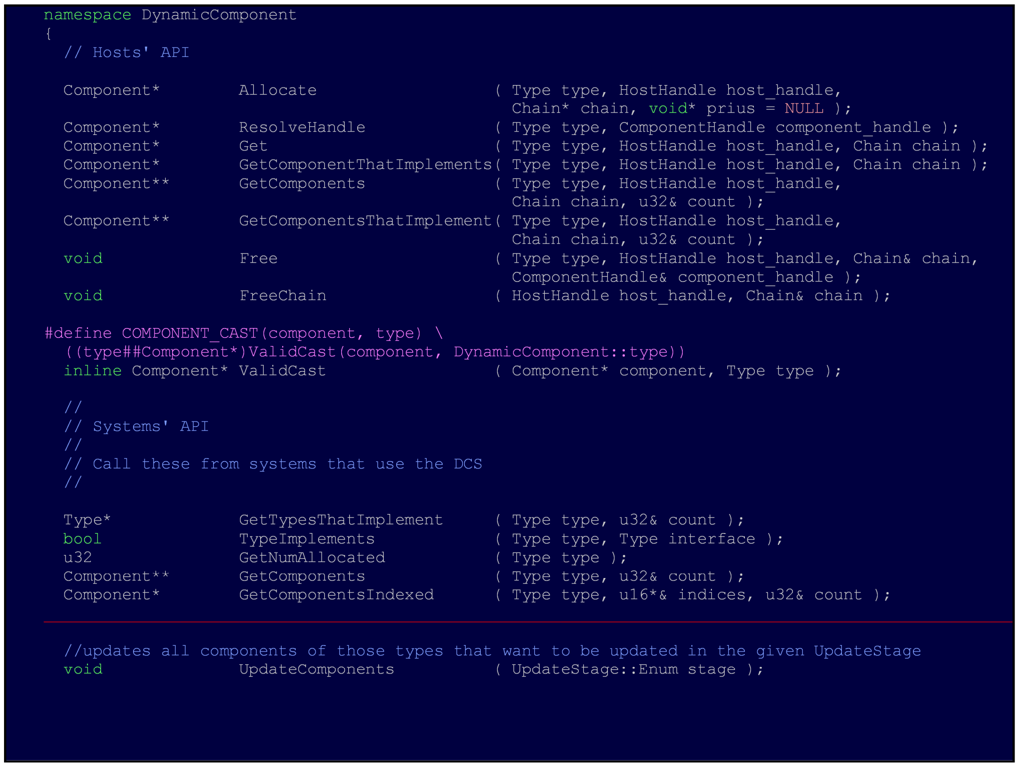
```
namespace DynamicComponent
{
  // Hosts' API

  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 );
  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*)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 );

  //returns whether component type implements interface
  bool              TypeImplements            ( Type type, Type interface );
```



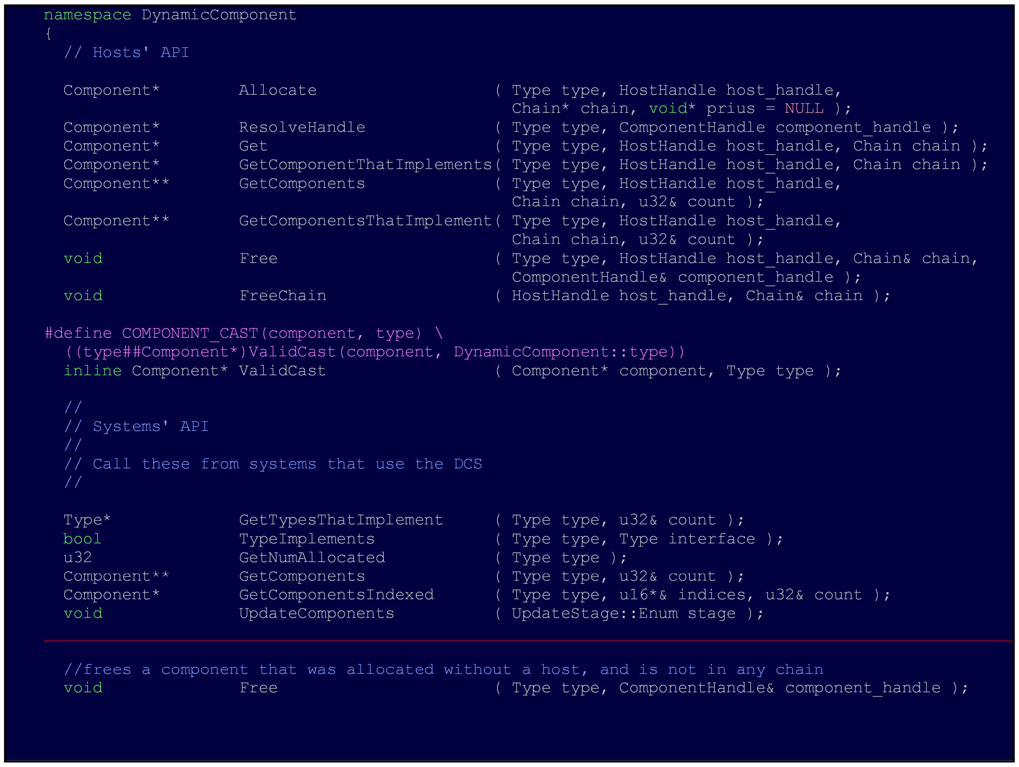
```
namespace DynamicComponent
{
  // Hosts' API

  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, 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*)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 );

  //returns the number of components of type that are currently allocated
  u32               GetNumAllocated           ( Type type );
```



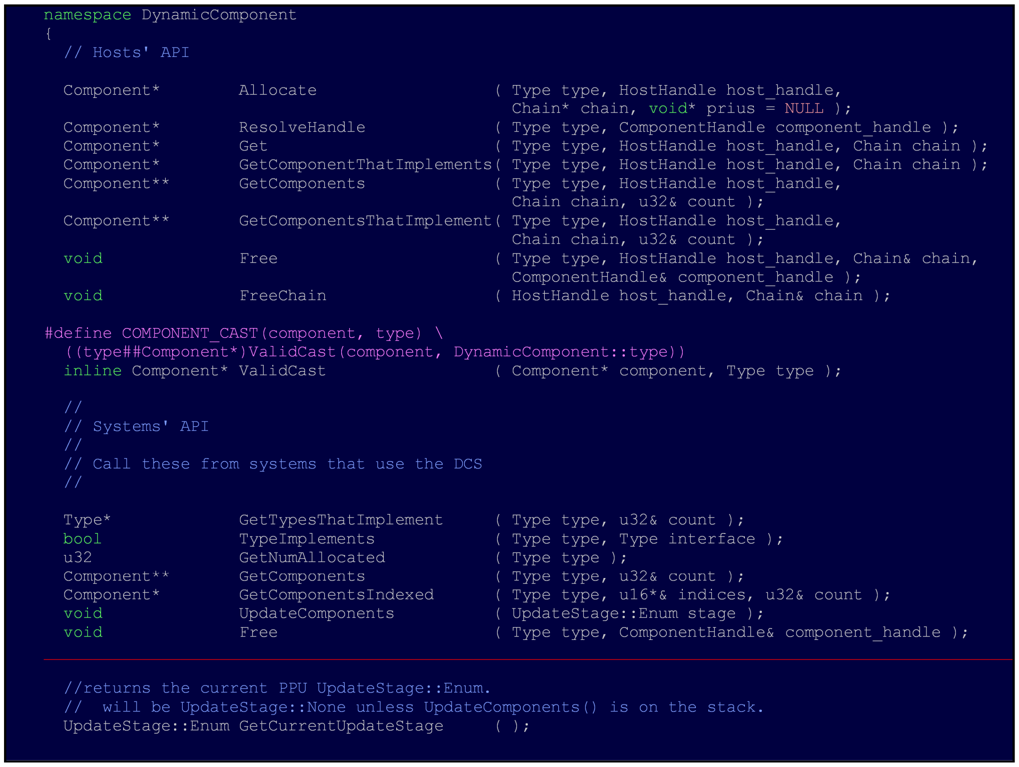
```cpp
namespace DynamicComponent
{
  // Hosts' API

  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*)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 );
```



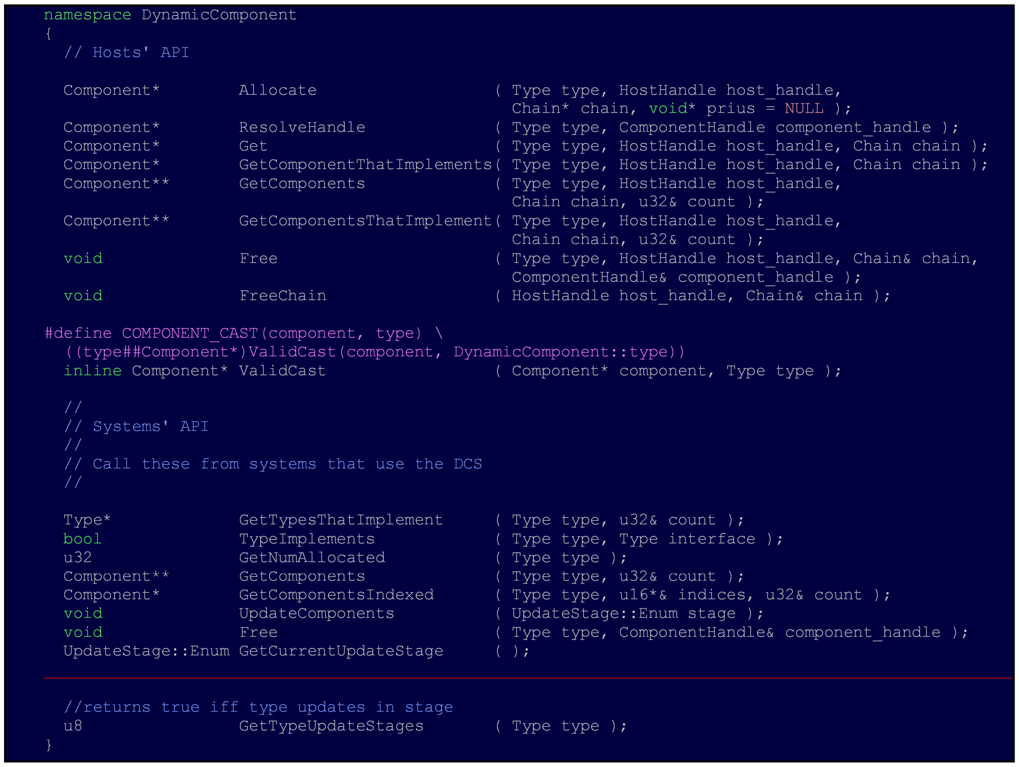
```
namespace DynamicComponent
{
  // Hosts' API

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

  //updates all components of those types that want to be updated in the given UpdateStage
  void              UpdateComponents          ( UpdateStage::Enum stage );
```



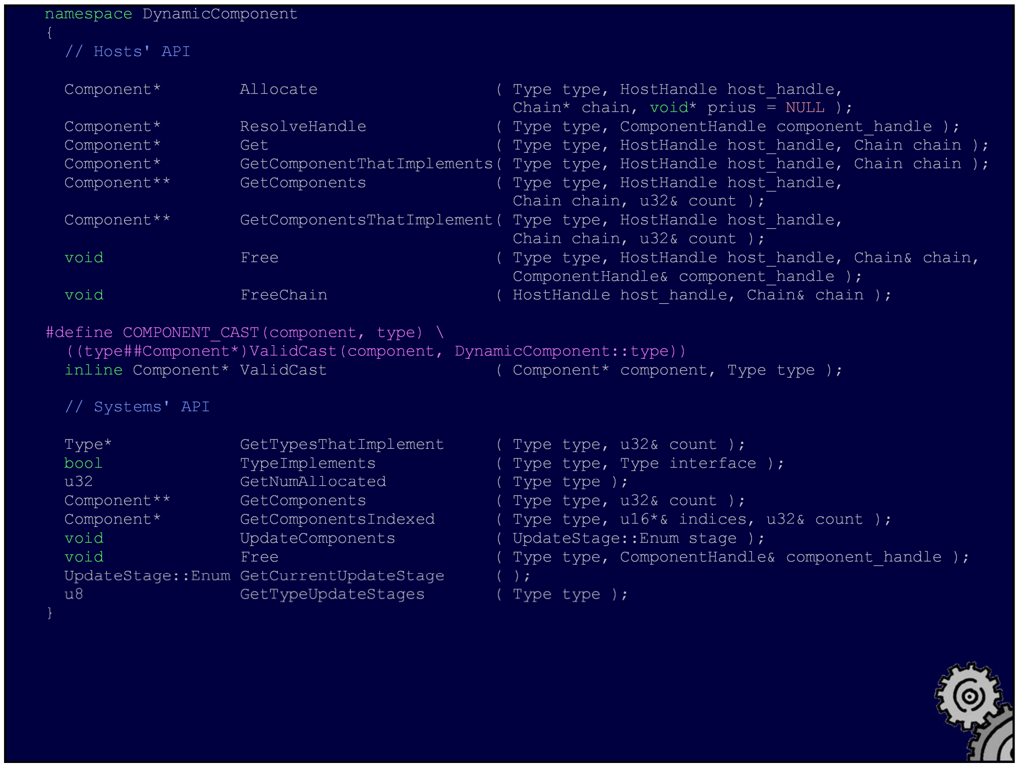
```cpp
namespace DynamicComponent
{
  // Hosts' API

  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*)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 );
  Component**       GetComponents             ( Type type, u32& count );
  Component*        GetComponentsIndexed      ( Type type, u16*& indices, u32& count );
  void              UpdateComponents          ( UpdateStage::Enum stage );

  //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,
                                                ComponentHandle& component_handle );
  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*             GetTypesThatImplement     ( Type type, u32& count );
  bool              TypeImplements            ( Type type, Type interface );
  u32               GetNumAllocated           ( Type type );
  Component**       GetComponents             ( Type type, u32& count );
  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     ( );
```



```
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*)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 );
  Component**       GetComponents             ( Type type, u32& count );
  Component*        GetComponentsIndexed      ( Type type, u16*& indices, u32& count );
  void              UpdateComponents          ( UpdateStage::Enum stage );
  void              Free                      ( Type type, ComponentHandle& component_handle );
  UpdateStage::Enum GetCurrentUpdateStage     ( );

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

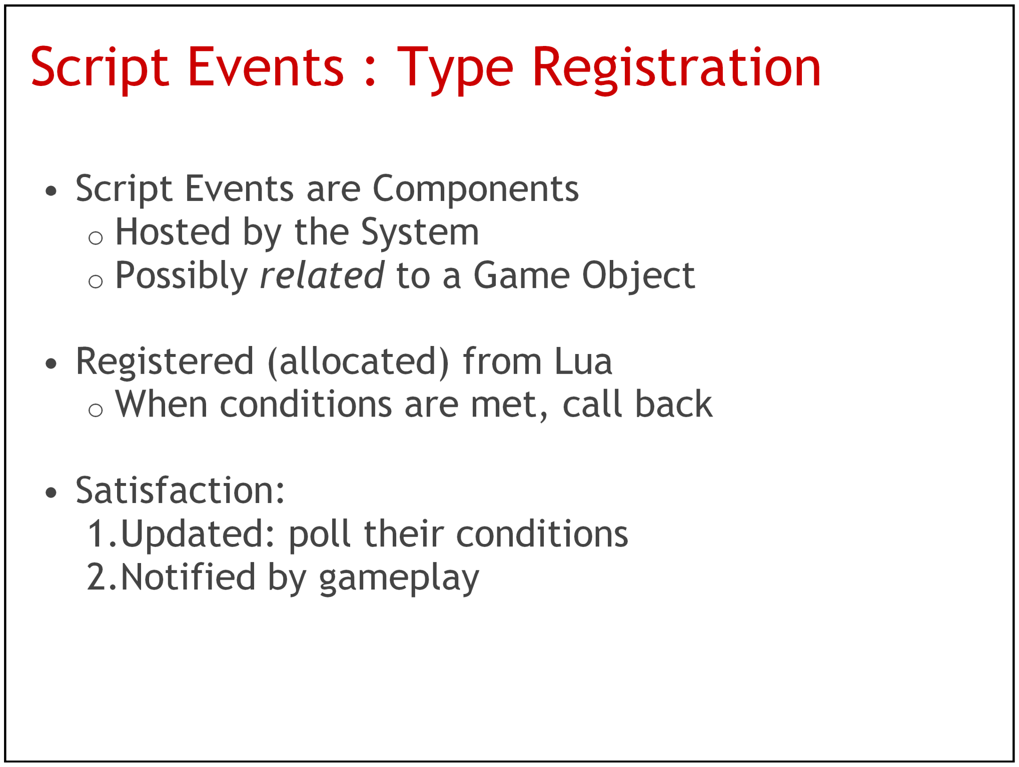


```
namespace DynamicComponent
{
  // Hosts' API

  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, Chain& chain );

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

  // Systems' API

  Type*             GetTypesThatImplement     ( Type type, u32& count );
  bool              TypeImplements            ( Type type, Type interface );
  u32               GetNumAllocated           ( Type type );
  Component**       GetComponents             ( Type type, u32& count );
  Component*        GetComponentsIndexed      ( Type type, u16*& indices, u32& count );
  void              UpdateComponents          ( UpdateStage::Enum stage );
  void              Free                      ( Type type, ComponentHandle& component_handle );
  UpdateStage::Enum GetCurrentUpdateStage     ( );
  u8                GetTypeUpdateStages       ( Type type );
}
```

GDC Canada
Game Developers Conference™ Canada
May 6-7, 2010
Vancouver Convention Centre | Vancouver, BC
UBM
REBOOT

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
  - Hosted by the System
  - Possibly *related* to a Game Object
- Registered (allocated) from Lua
  - When conditions are met, call back
- Satisfaction:
  1. Updated: poll their conditions
  2. Notified by gameplay

x

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

- 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

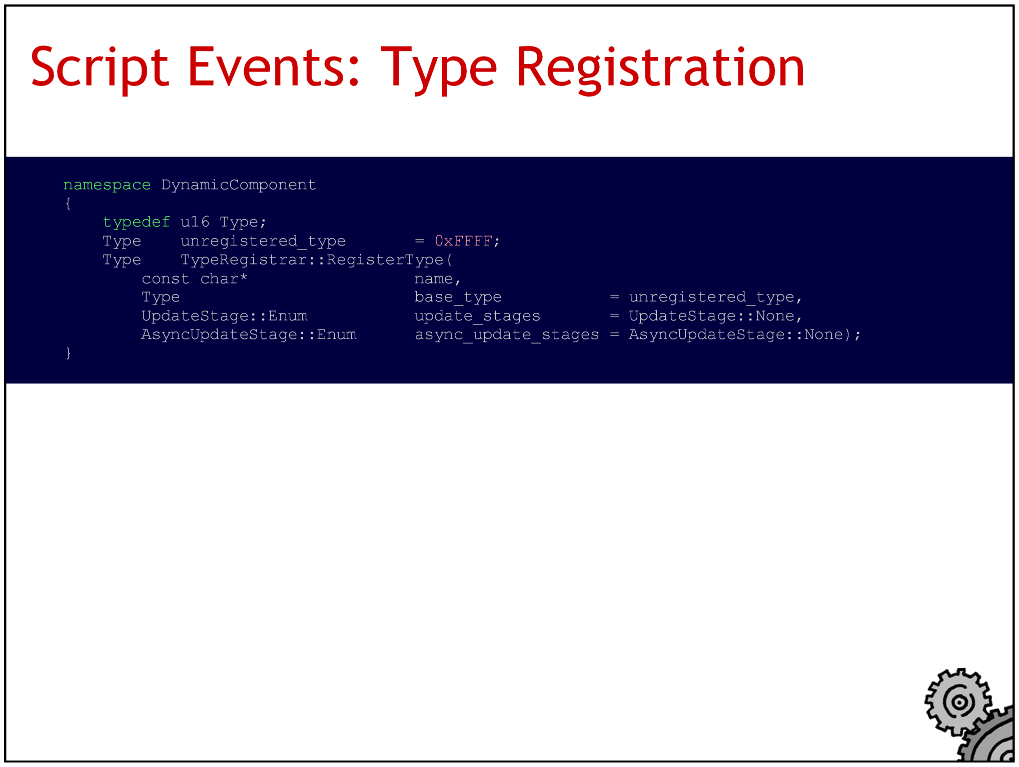
Script Events : Type Registration

Notified

- Checkpoint
- Damage
- Death
- NumAlive
- Active
- Custom
  - Reload
  - Zoom
  - Grapple
  - SeatEnter

Updated

- Location
- Timer
- Visible



Script Events: Type Registration

```
namespace DynamicComponent
{
    typedef u16 Type;
    Type    unregistered_type       = 0xFFFF;
    Type    TypeRegistrar::RegisterType(
        const char*                 name,
        Type                        base_type           = unregistered_type,
        UpdateStage::Enum           update_stages       = UpdateStage::None,
        AsyncUpdateStage::Enum      async_update_stages = AsyncUpdateStage::None);
}
```

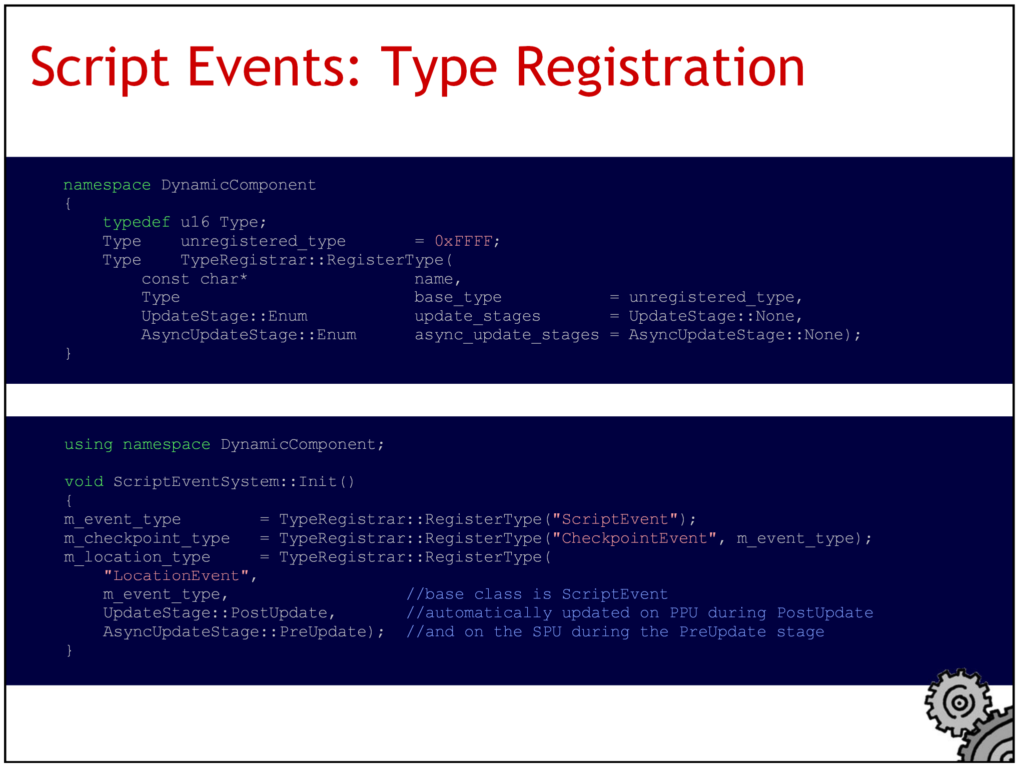
Types are registered through the DynamicComponent::TypeRegistrar.

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


Script Events: Type Registration

```
namespace DynamicComponent
{
    typedef u16 Type;
    Type    unregistered_type       = 0xFFFF;
    Type    TypeRegistrar::RegisterType(
        const char*                 name,
        Type                        base_type           = unregistered_type,
        UpdateStage::Enum           update_stages       = UpdateStage::None,
        AsyncUpdateStage::Enum      async_update_stages = AsyncUpdateStage::None);
}
```

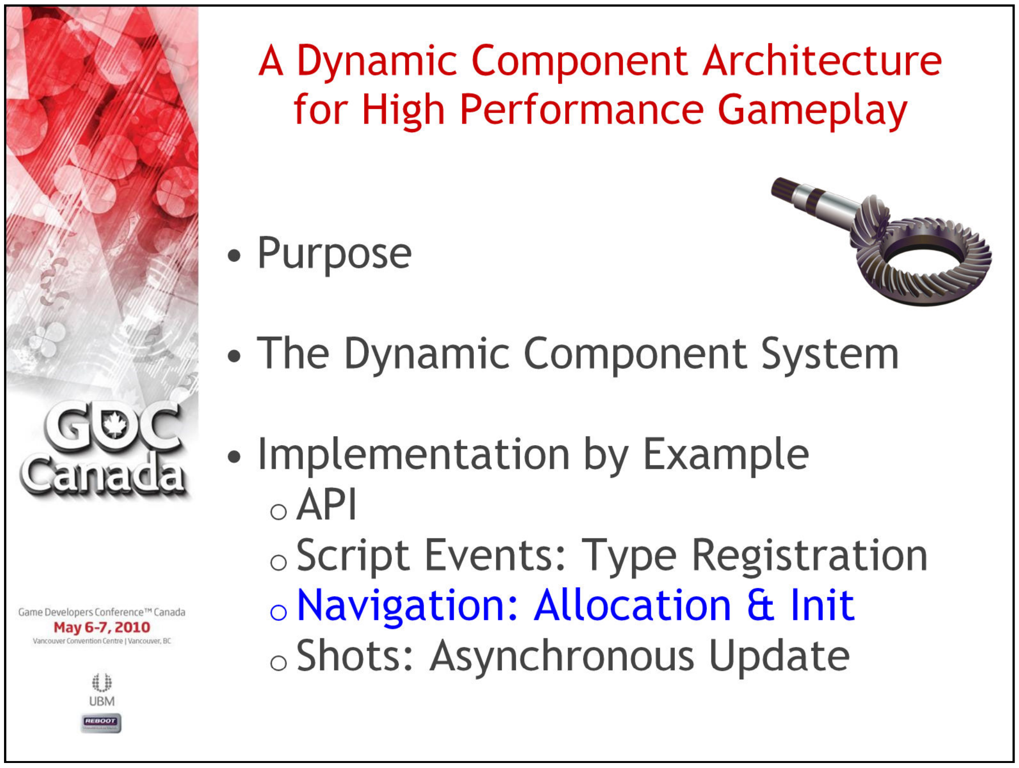
```
using namespace DynamicComponent;

void ScriptEventSystem::Init()
{
m_event_type        = TypeRegistrar::RegisterType("ScriptEvent");
m_checkpoint_type   = TypeRegistrar::RegisterType("CheckpointEvent", m_event_type);
m_location_type     = TypeRegistrar::RegisterType(
    "LocationEvent",
    m_event_type,                   //base class is ScriptEvent
    UpdateStage::PostUpdate,        //automatically updated on PPU during PostUpdate
    AsyncUpdateStage::PreUpdate);   //and on the SPU during the PreUpdate stage
}
```

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



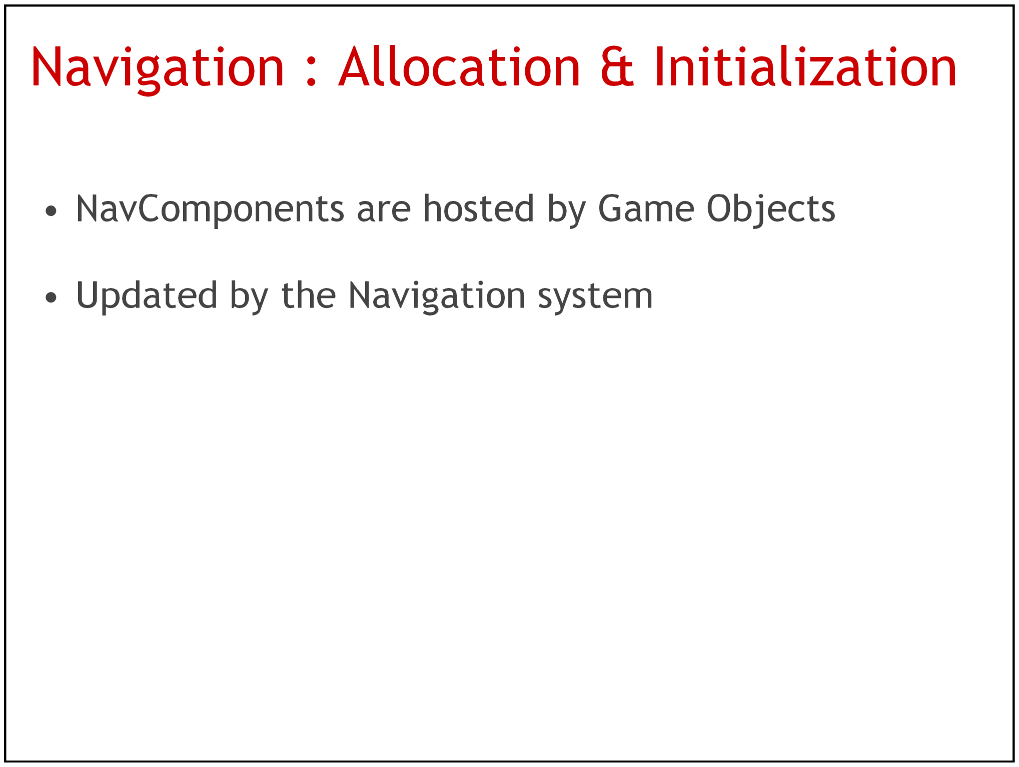
A Dynamic Component Architecture for High Performance Gameplay

- Purpose
- The Dynamic Component System
- Implementation by Example
  - API
  - Script Events: Type Registration
  - Navigation: Allocation & Init
  - Shots: Asynchronous Update

GDC Canada
Game Developers Conference™ Canada
May 6-7, 2010
Vancouver Convention Centre | Vancouver, BC
UBM
REBOOT

- At a high level, I have three sections to this discussion

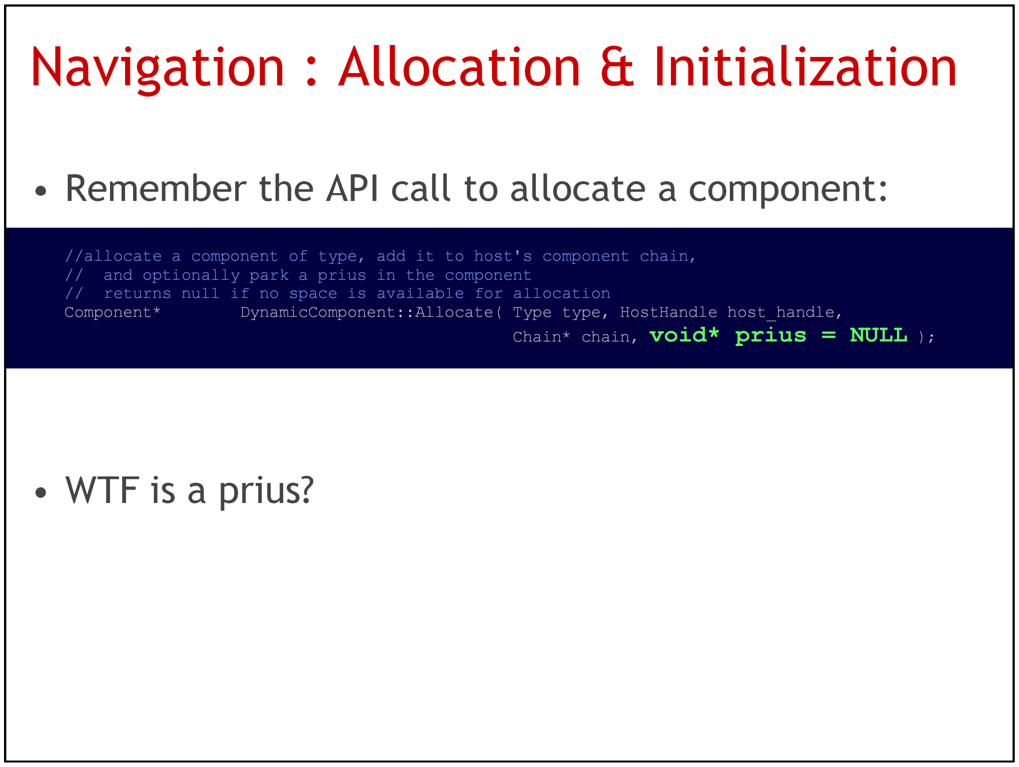
1. First, we'll talk about our purpose here.
   - We'll discuss the problem we're trying to solve, and my approach to a solution
2. Second, we'll talk about details of my solution, which I call the Dynamic Component System.
   - We'll discuss features of the system, and why it's a good solution
3. And 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



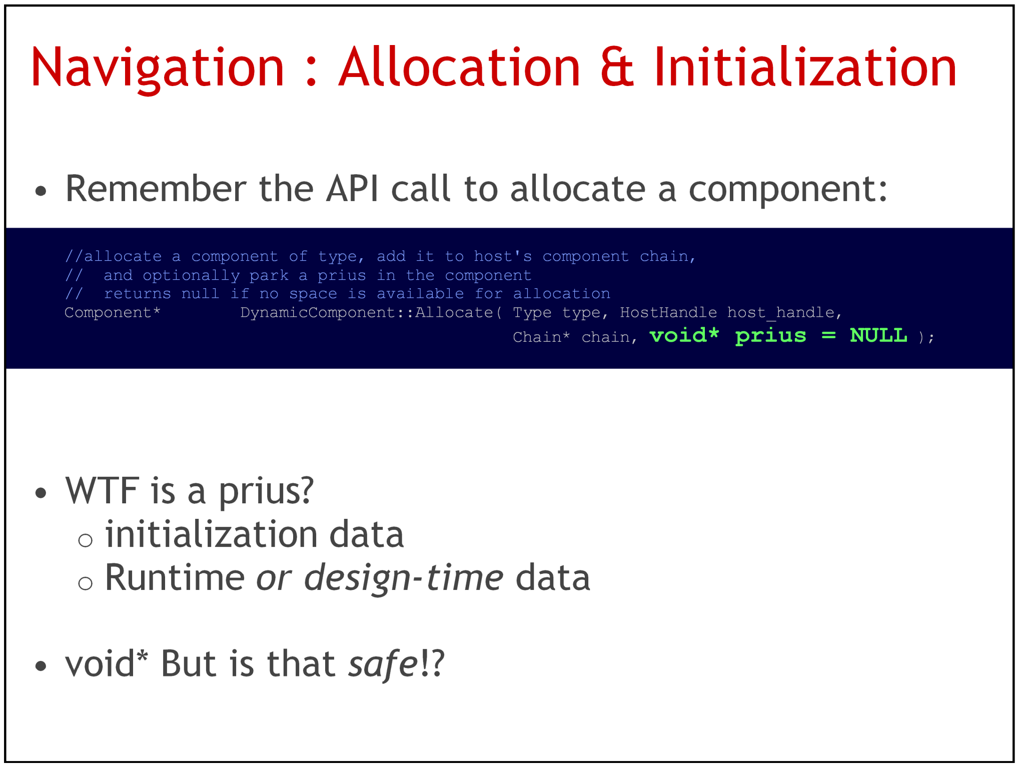
Navigation : Allocation & Initialization

- Remember the API call to allocate a component:

```
//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?

- 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!

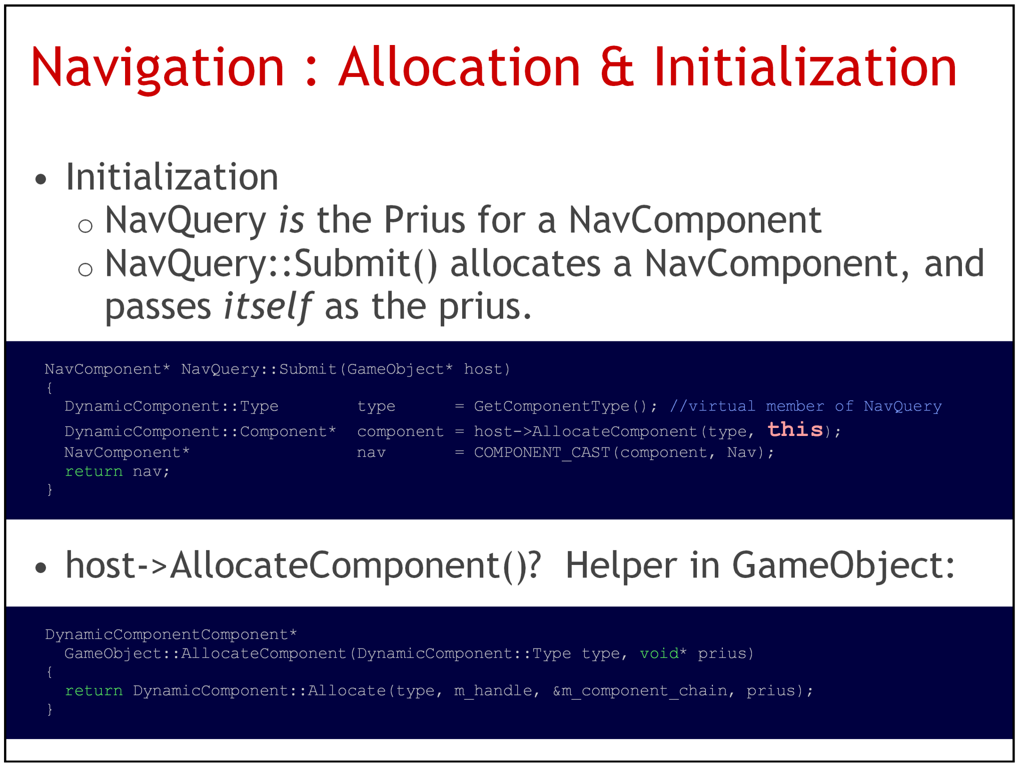

Navigation : Allocation & Initialization

- Remember the API call to allocate a component:

```
//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!



Navigation : Allocation & Initialization

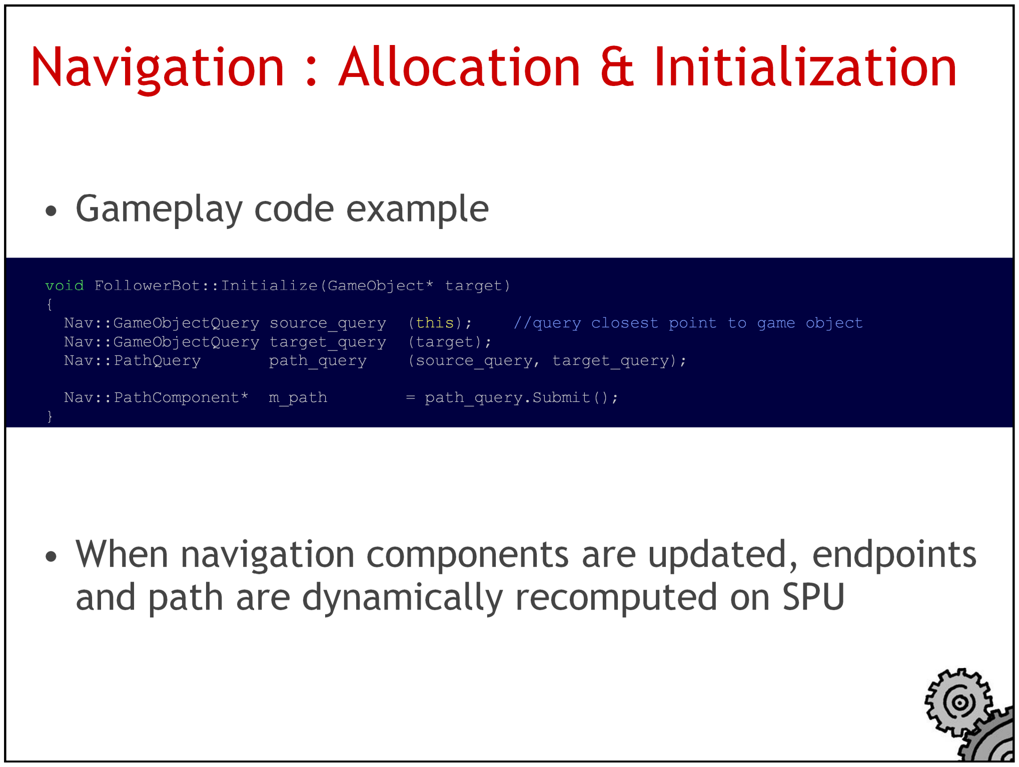
- Initialization
  - NavQuery *is* the Prius for a NavComponent
  - NavQuery::Submit() allocates a NavComponent, and passes *itself* as the prius.

```
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:

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

- Interesting initialization idiom
  - A NavQuery is the Prius for a NavComponent
  - NavQuery::Submit() allocates a NavComponent, and passes itself as the prius.



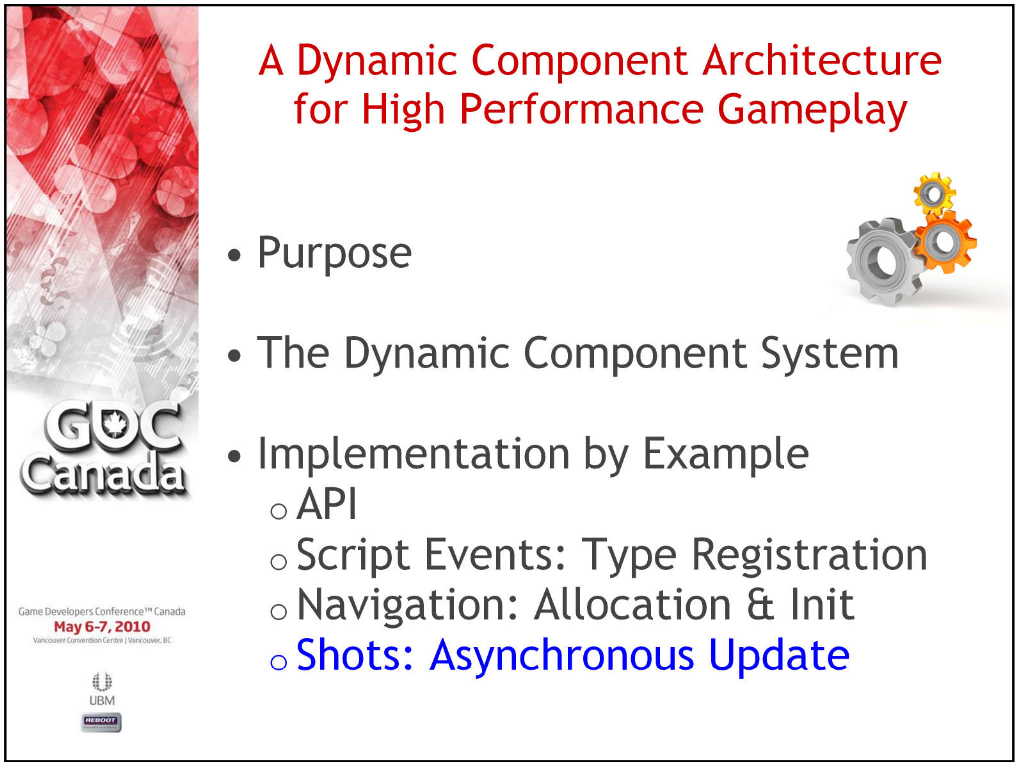
Navigation : Allocation & Initialization

- Gameplay code example

```
void FollowerBot::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*  m_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.



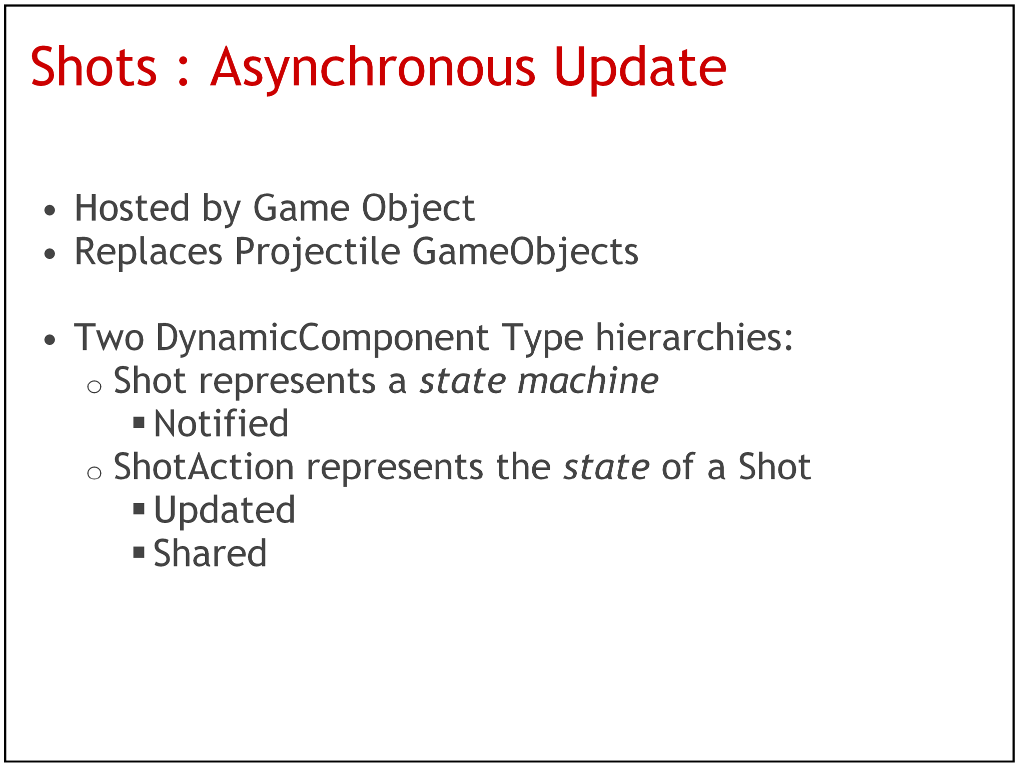
A Dynamic Component Architecture for High Performance Gameplay

- Purpose
- The Dynamic Component System
- Implementation by Example
  - API
  - Script Events: Type Registration
  - Navigation: Allocation & Init
  - Shots: Asynchronous Update

Game Developers Conference™ Canada
May 6-7, 2010
Vancouver Convention Centre | Vancouver, BC

- 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



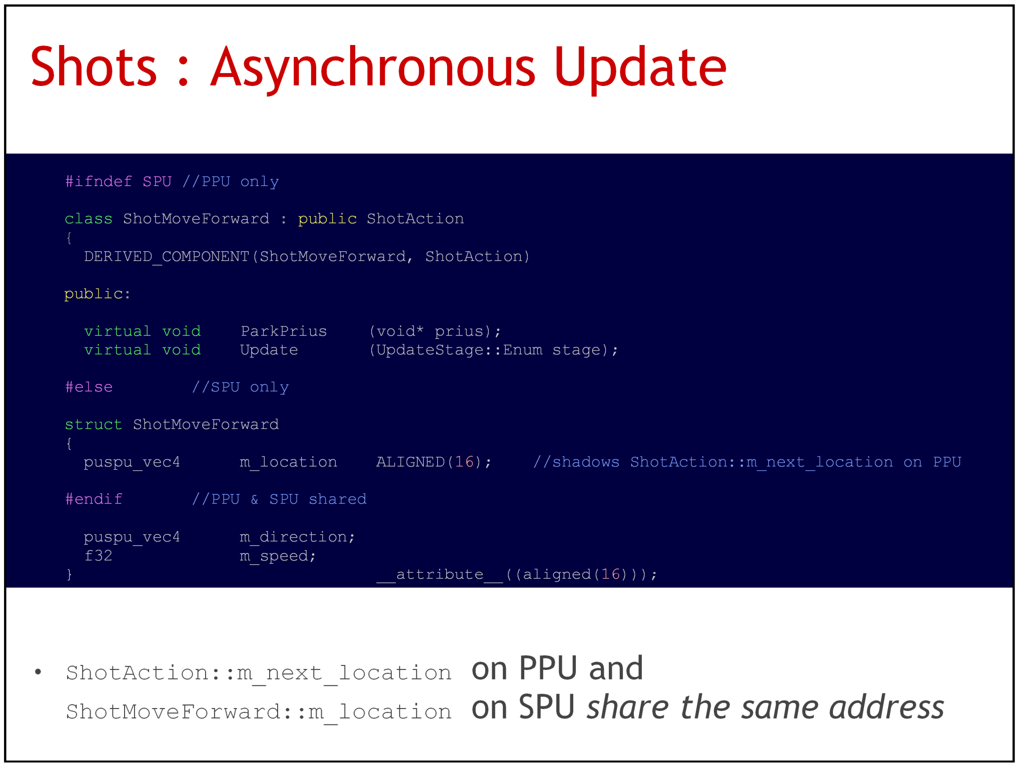
Shots : Asynchronous Update

- Hosted by Game Object
- Replaces Projectile GameObjects

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

x

- 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

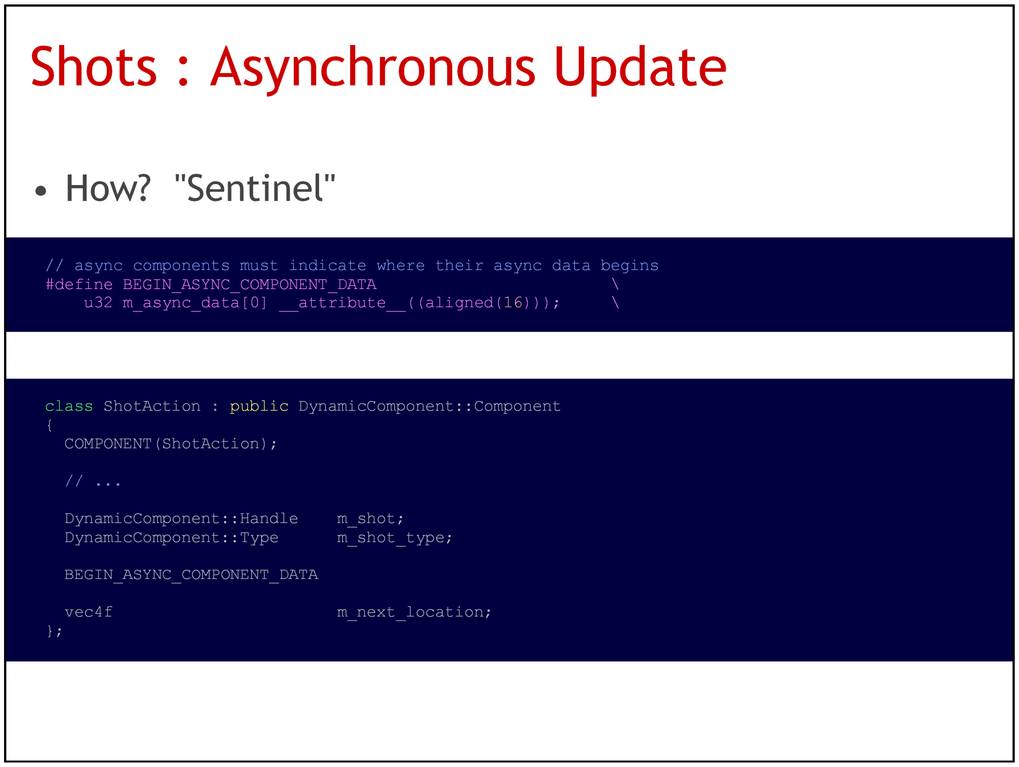


Shots : Asynchronous Update

```
#ifndef SPU //PPU only

class ShotMoveForward : public ShotAction
{
  DERIVED_COMPONENT(ShotMoveForward, ShotAction)

public:

  virtual void    ParkPrius    (void* prius);
  virtual void    Update       (UpdateStage::Enum stage);

#else        //SPU only

struct ShotMoveForward
{
  puspu_vec4      m_location    ALIGNED(16);    //shadows ShotAction::m_next_location on PPU

#endif       //PPU & SPU shared

  puspu_vec4      m_direction;
  f32             m_speed;
}                               __attribute__((aligned(16)));
```

- `ShotAction::m_next_location` on PPU and `ShotMoveForward::m_location` on SPU *share the same address*

- 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.



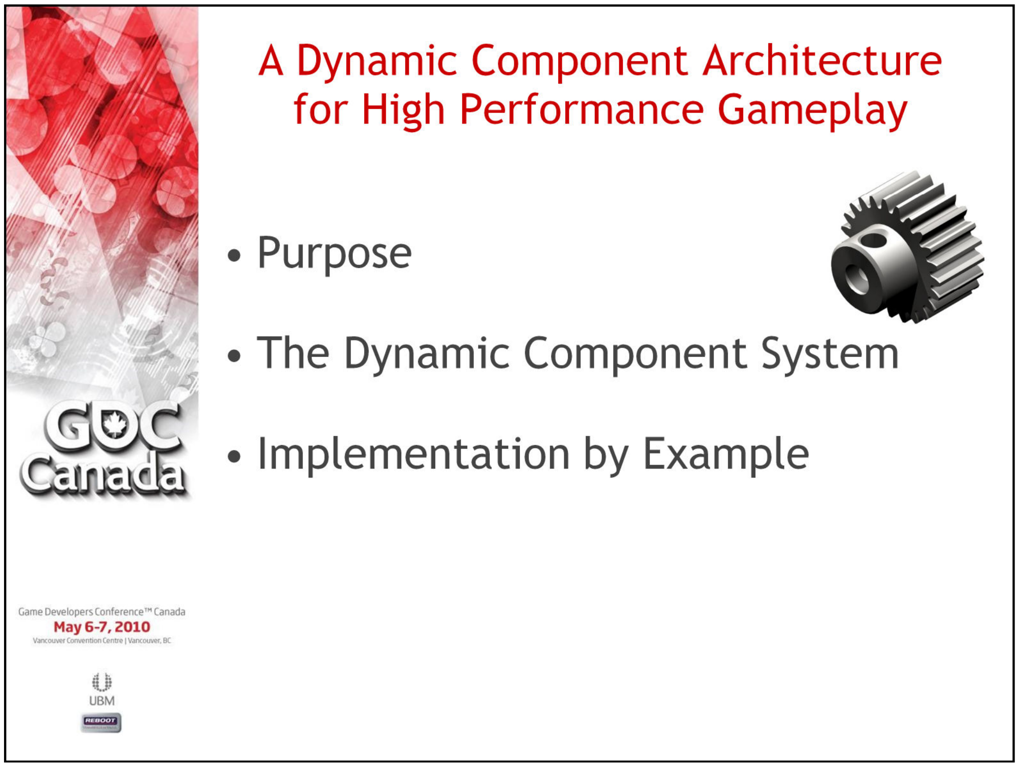
Shots : Asynchronous Update

- How? "Sentinel"

```
// async components must indicate where their async data begins
#define BEGIN_ASYNC_COMPONENT_DATA                              \
    u32 m_async_data[0] __attribute__((aligned(16)));      \
```

```
class ShotAction : public DynamicComponent::Component
{
  COMPONENT(ShotAction);

  // ...

  DynamicComponent::Handle    m_shot;
  DynamicComponent::Type      m_shot_type;

  BEGIN_ASYNC_COMPONENT_DATA

  vec4f                       m_next_location;
};
```

- "Sentinel" data member identifies start of SPU instance
  - 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.
  - The sentinel is a zero-sized array member



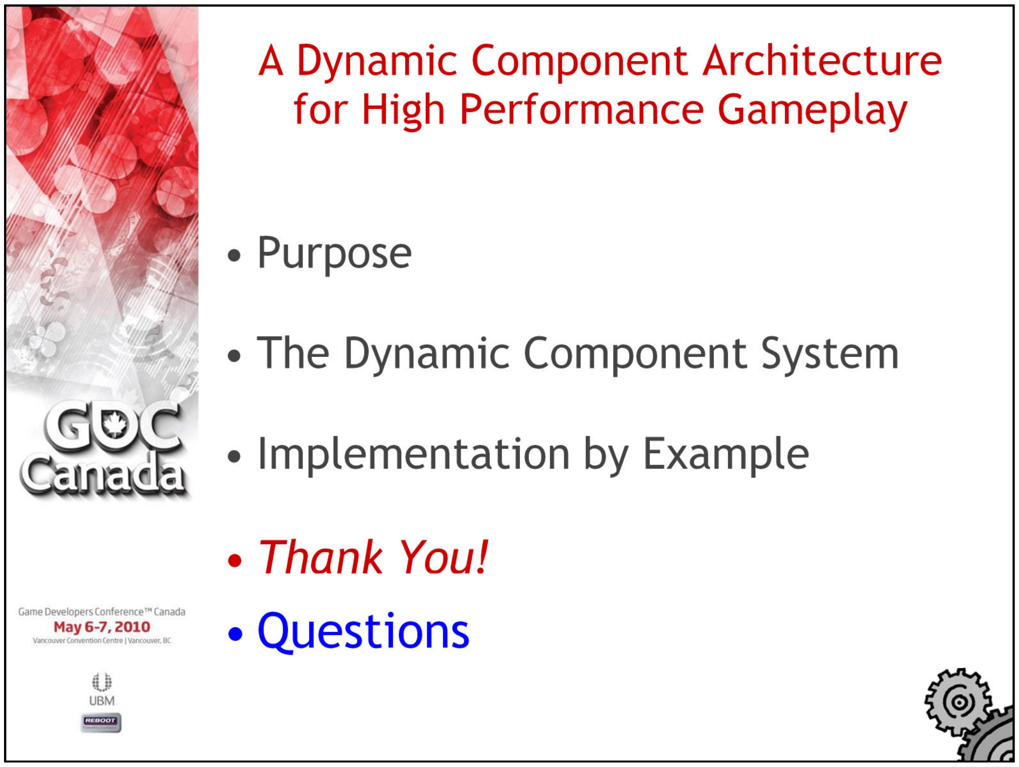
A Dynamic Component Architecture for High Performance Gameplay

- Purpose
- The Dynamic Component System
- Implementation by Example

Game Developers Conference™ Canada
May 6-7, 2010
Vancouver Convention Centre | Vancouver, BC

What we've discussed

1. First, we talked about our purpose here.
    - We discussed the problem we wre 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



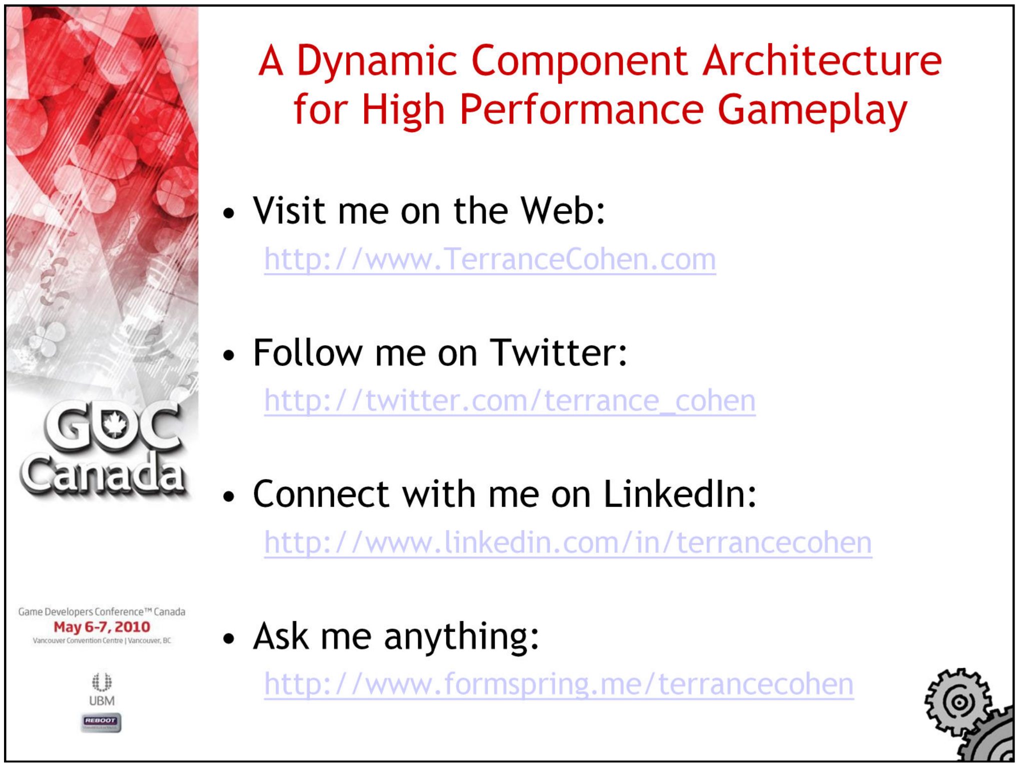
A Dynamic Component Architecture for High Performance Gameplay

- Purpose
- The Dynamic Component System
- Implementation by Example
- *Thank You!*
- Questions

Game Developers Conference™ Canada
May 6-7, 2010
Vancouver Convention Centre | Vancouver, BC

What we've discussed

1. First, we talked about our purpose here.
   - We discussed the problem we wre 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



A Dynamic Component Architecture for High Performance Gameplay

- 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 anything:
  http://www.formspring.me/terrancecohen

GDC Canada
Game Developers Conference™ Canada
May 6-7, 2010
Vancouver Convention Centre | Vancouver, BC
UBM
REBOOT

What we've discussed

1. First, we talked about our purpose here.
   - We discussed the problem we wre 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

UBM

REBOOT

INSOMNIAC GAMES

A Dynamic Component Architecture for High Performance Gameplay

Terrance Cohen
Lead Systems Engineer
Insomniac Games

www.TerranceCohen.com
Twitter: @terrance_cohen

Game Developers Conference™ Canada
May 6-7, 2010
Vancouver Convention Centre | Vancouver, BC
www.GDC-Canada.com

GDC Canada

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