



# SPU gameplay

Joe Valenzuela

[joe@insomniacgames.com](mailto:joe@insomniacgames.com)

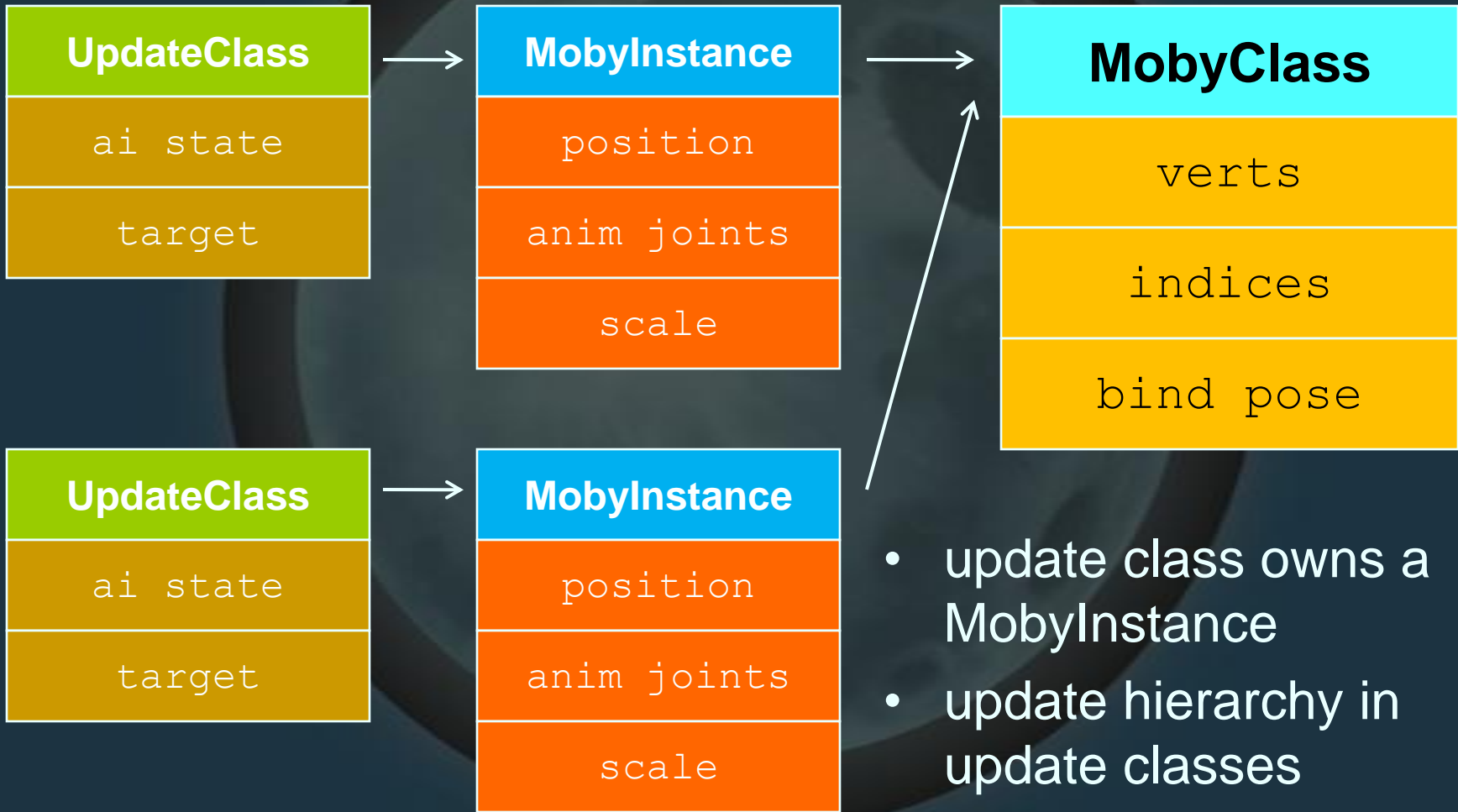
GDC 2009

# glossary

- mobys
  - class
  - instances
- update classes
- AsyncMobyUpdate
  - Guppys
  - Async
- aggregateupdate



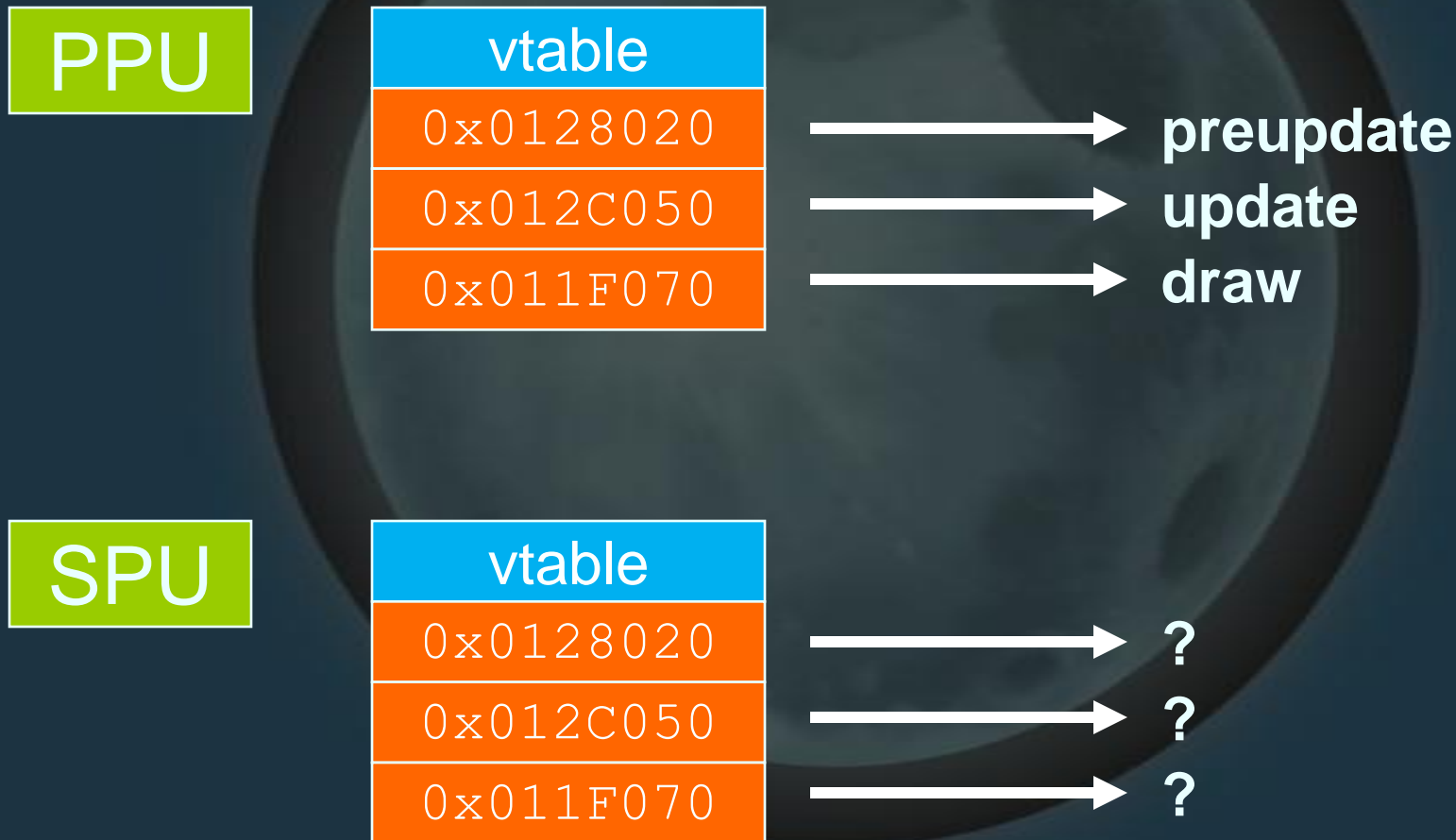
# Moby?



# spu gameplay difficulties

- multiprocessor
- NUMA
- different ISA
- it's different
  - takes time and effort to retrofit code
  - unfamiliarity with the necessary upfront design

# your virtual functions don't work



# your pointers don't work

```
struct foo_t
{
    float m_t;
    float m_scale;
    u32    m_flags;
    u16*   m_points;
};
```



# your code doesn't compile

```
x:/core/code/users/jvalenzu/shared/igCore/igsys/igDebug.h(19,19): error:
  libsn.h: No such file or directory
x:/core/code/users/jvalenzu/shared/igCore/igTime/igTimer.h(15,27): error:
  sys/time_util.h: No such file or directory
pickup/pickupbase_preupdate_raw.inc(160): error: 'DEFAULT_FLAGS' is not a
  member of 'COLL'
pickup/pickupbase_preupdate_raw.inc(160): error: 'EXCLUDE_HERO_ONLY' is
  not a member of 'COLL'
x:/core/code/users/jvalenzu/shared/igCore/igPhysics/ppu/igPhysics.h(293):
  error: expected unqualified-id before '*' token
x:/core/code/users/jvalenzu/shared/igCore/igPhysics/ppu/igPhysics.h(293):
  error: expected ',', or '...' before '*' token
x:/core/code/users/jvalenzu/shared/igCore/igPhysics/ppu/igPhysics.h(293):
  error: ISO C++ forbids declaration of 'parameter' with no type
x:/core/code/users/jvalenzu/shared/igCore/igg/igShaderStructs.h: At global
  scope:
x:/core/code/users/jvalenzu/shared/igCore/igg/igShaderStructs.h(22):
  error: redefinition of 'struct VtxVec4'
x:/core/code/users/jvalenzu/shared/igCore/igsys/igTypes.h(118): error:
  previous definition of 'struct VtxVec4'
```

# object driven update

```
for(i = 0; i < num_entities; ++i) {  
    entity* e = &g_entity_base[i];  
  
    e->collect_info();  
    e->update();  
  
    e->move();  
    e->animate();  
    e->etc();  
}
```

- can't amortize setup costs
- can't hide much deferred work

# more modular update

1

```
for(i = 0, e = &g_entity_base[0]; i < num_ent; ++i, ++e) {  
    e->collect_info();  
    e->issue_anim_request();  
}
```

```
for(i = 0, e = &g_entity_base[0]; i < num_ent; ++i, ++e)  
    e->update();
```

2

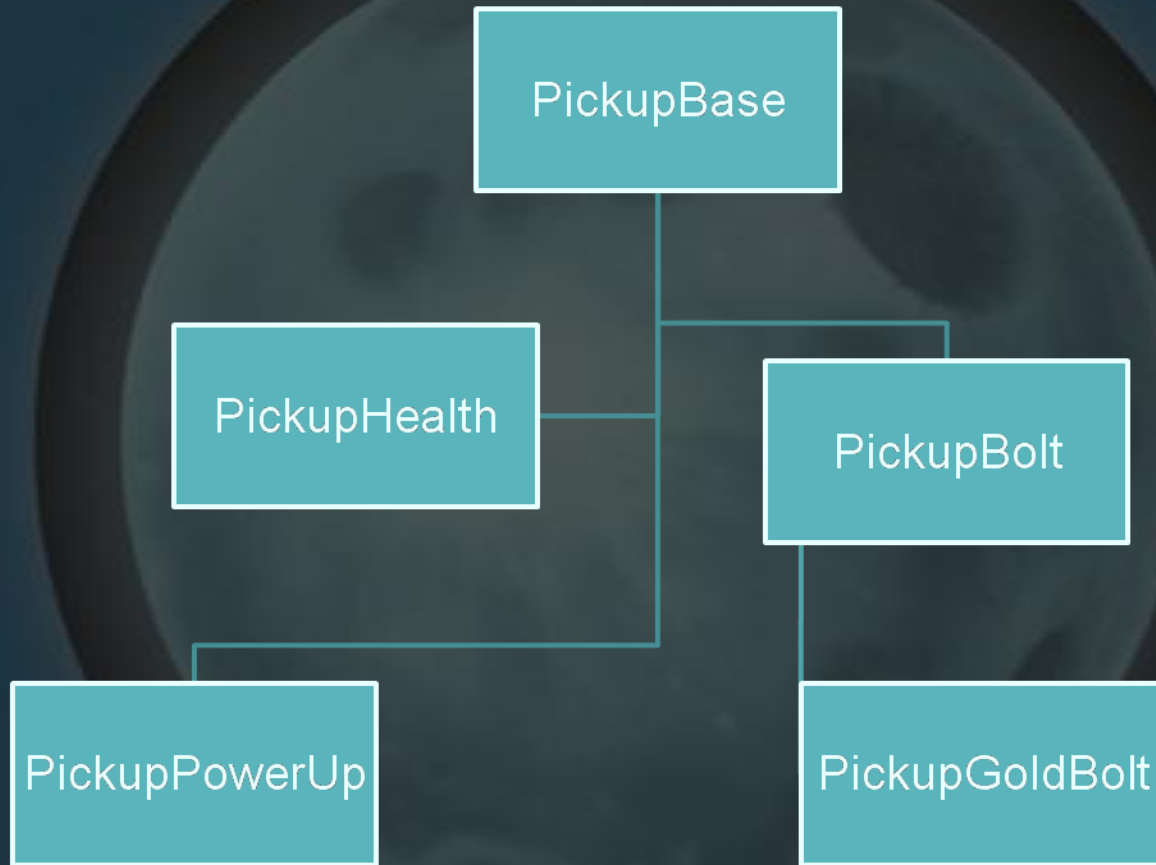
```
finalize_animation();
```

```
for(i = 0, e = &g_entity_base[0]; i < num_ent; ++i, ++e)  
    e->postupdate();
```

# aggregate updating

- group instances by type
  - further sort each group to minimize state change
- one aggregate updater per type, with multiple code fragments
- combined ppu & spu update
- more opportunity to amortize cost of expensive setup

# aggregate example (pickup)



# aggregate example cont...

## Pickup Instances

PickupBolt

PickupBolt

PickupHealth

PickupHealth

PickupHealth

`pickupbolt_preupdate`  
`pickupbolt_update`

`pickuphealth_preupdate`  
`pickuphealth_update`  
`pickuphealth_postupdate`

# a trivial optimization

```
void TruckUpdate::Update()
{
    if(m_wait_frame > TIME::GetCurrentFrame())
    {
        return;
    }

    // ... more work
}
```

# a trivial optimization

```
void TruckUpdate::Update()
{
    if(m_wait_frame > TIME::GetCurrentFrame())
    {
        return;
    }

    // ... more work
}
```

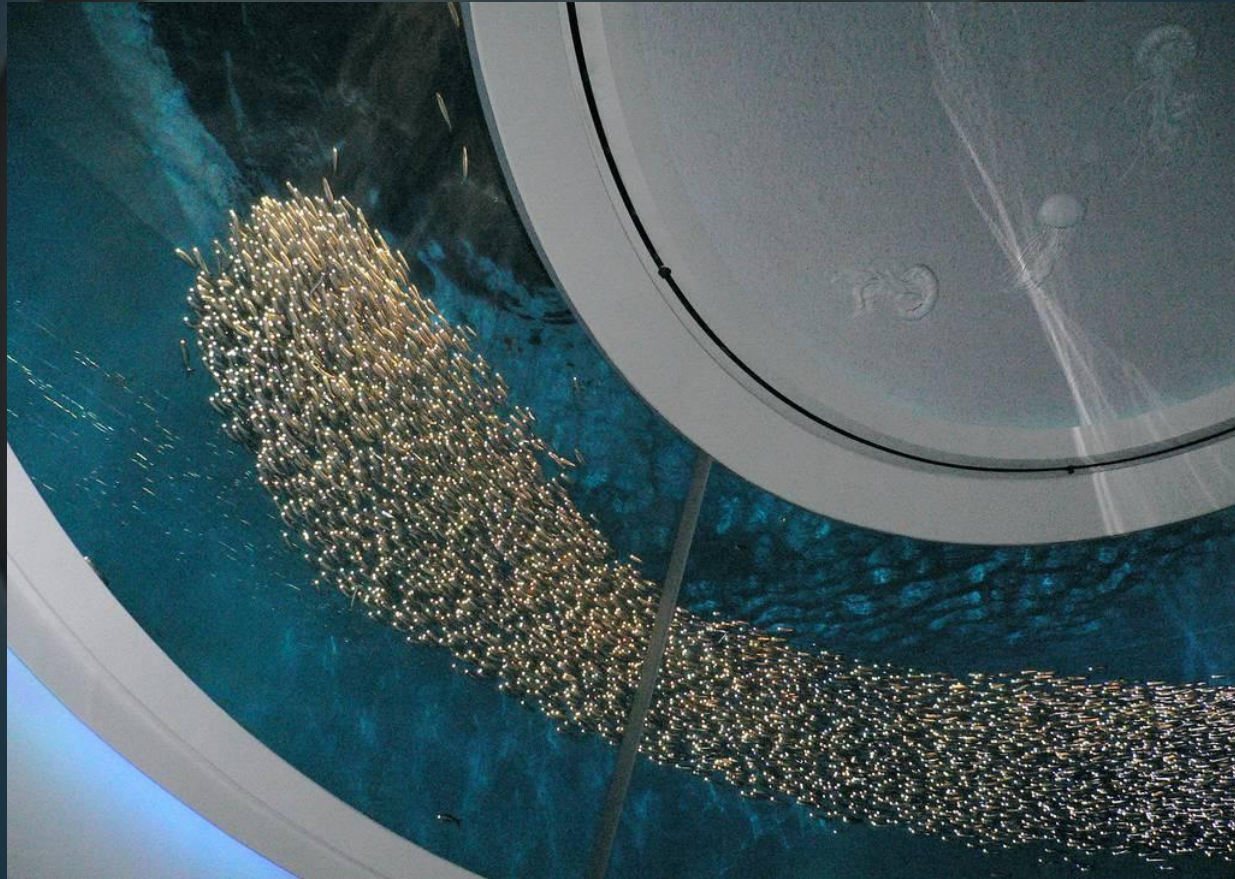
# a trivial optimization (cont)

```
void Aggregate_TruckUpdate_Update()
{
    u32 current_frame = TIME::GetCurrentFrame();

    for(u32 i = 0; i < m_count; ++i)
    {
        TruckUpdate* self = &m_updates[i];
        if(self->m_wait_frame > current_frame)
        {
            continue;
        }

        // ... more work
    }
}
```

# SPU gameplay systems



# SPU gameplay intro

- systems built around applying shaders to lots of homogenous data
  - AsyncMobyUpdate
  - Guppys
  - AsyncEffect
- small, simple code overlays
  - user-supplied
  - compiled offline
  - debuggable
  - analogous to graphics shaders

# async moby update overview



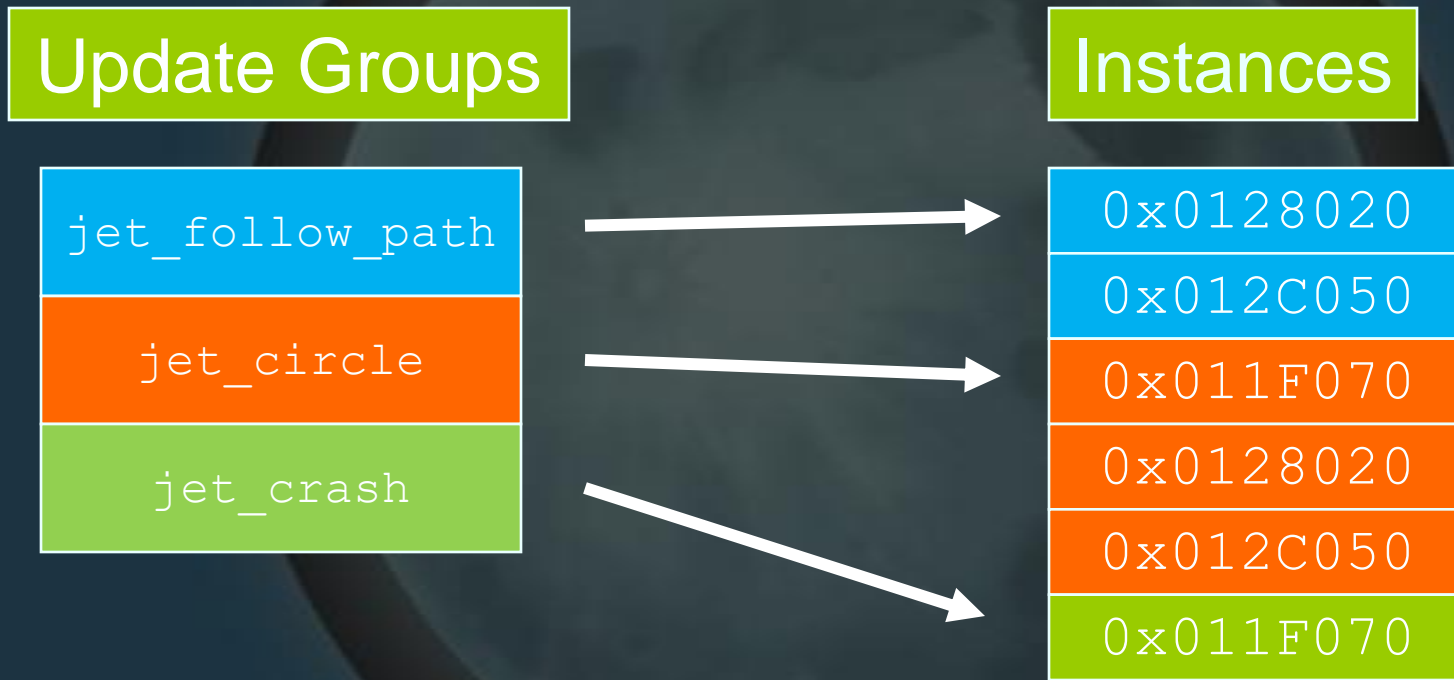
# overview

- AsyncMobyUpdate
  - base framework, meant to work with update classes
  - retains MobyInstance rendering pipeline
- Guppys
  - “light” MobyInstance replacement
  - 100% SPU update, no update class
  - 90% MobyInstance rendering pipeline
- AsyncEffect
  - very easy fire & forget SPU “effects”
  - user-selectable, not user-written, shaders

# async moby update

- designed to move update classes to SPU
- user supplied update routine in code fragment
- multiple code fragments per update class
  - one per AI state, for example
- user-defined instance data format
- user-defined common data
- extern code provided through function pointer tables

# async moby update (cont...)



- update group per code fragment
- instances bound to update group each frame

# instance vs common

- instance data
  - data transformed by your update routine
  - *e.g.* a Jet, or zombie limb
- common data
  - data common to all instances of the same type
  - *e.g.* class-static variables, current frame

# function pointer table interface

```
struct global_funcs_t
{
    void (*print) (const char *fmt, ...);
    // ...
    f32  (*get_current_time) ();
    u32  (*read_decrementer) ();
    u32  (*coll_swept_sphere) (qword p0, qword p1, u32 flags);
    void (*coll_get_result)  (COLL::Result *dest, u32 id, u32 tag);
};
```

- debug, print functions
- access to common data, timestep
- collision & FX routines

# simple API

## setup:

```
u32 tag = AsyncMobyUpdate::AllocTag();

AsyncMobyUpdate::RegisterType      (tag, truck_frag_start, truck_frag_size);

// common setup
AsyncMobyUpdate::SetNumCommonBlocks(tag, 1);
AsyncMobyUpdate::SetCommonBlock    (tag, 0,
                                     &g_TruckCommon, sizeof g_TruckCommon);

// instance setup
AsyncMobyUpdate::SetNumInstanceStreams(tag, 1);
AsyncMobyUpdate::SetOffset          (tag, 0, 0);
AsyncMobyUpdate::SetStride          (tag, 0, sizeof(TruckClass));
```

## use:

```
AsyncMobyUpdate::AddInstances      (tag, instance_block, count);
```

# allocate tag

## setup:

```
u32 tag = AsyncMobyUpdate::AllocTag();

AsyncMobyUpdate::RegisterType      (tag, truck_frag_start, truck_frag_size);

// common setup
AsyncMobyUpdate::SetNumCommonBlocks(tag, 1);
AsyncMobyUpdate::SetCommonBlock    (tag, 0,
                                     &g_TruckCommon, sizeof g_TruckCommon);

// instance setup
AsyncMobyUpdate::SetNumInstanceStreams(tag, 1);
AsyncMobyUpdate::SetOffset          (tag, 0, 0);
AsyncMobyUpdate::SetStride          (tag, 0, sizeof(TruckClass));
```

## use:

```
AsyncMobyUpdate::AddInstances      (tag, instance_block, count);
```

# register fragment

## setup:

```
u32 tag = AsyncMobyUpdate::AllocTag();

AsyncMobyUpdate::RegisterType      (tag, truck_frag_start, truck_frag_size);

// common setup
AsyncMobyUpdate::SetNumCommonBlocks(tag, 1);
AsyncMobyUpdate::SetCommonBlock    (tag, 0,
                                     &g_TruckCommon, sizeof g_TruckCommon);

// instance setup
AsyncMobyUpdate::SetNumInstanceStreams(tag, 1);
AsyncMobyUpdate::SetOffset          (tag, 0, 0);
AsyncMobyUpdate::SetStride          (tag, 0, sizeof(TruckClass));
```

## use:

```
AsyncMobyUpdate::AddInstances      (tag, instance_block, count);
```

# set common block info

setup:

```
u32 tag = AsyncMobyUpdate::AllocTag();

AsyncMobyUpdate::RegisterType      (tag, truck_frag_start, truck_frag_size);

// common setup
AsyncMobyUpdate::SetNumCommonBlocks (tag, 1);
AsyncMobyUpdate::SetCommonBlock     (tag, 0,
                                     &g_TruckCommon, sizeof g_TruckCommon);

// instance setup
AsyncMobyUpdate::SetNumInstanceStreams (tag, 1);
AsyncMobyUpdate::SetOffset             (tag, 0, 0);
AsyncMobyUpdate::SetStride             (tag, 0, sizeof(TruckClass));
```

use:

```
AsyncMobyUpdate::AddInstances      (tag, instance_block, count);
```

# set instances info

## setup:

```
u32 tag = AsyncMobyUpdate::AllocTag();

AsyncMobyUpdate::RegisterType      (tag, truck_frag_start, truck_frag_size);

// common setup
AsyncMobyUpdate::SetNumCommonBlocks(tag, 1);
AsyncMobyUpdate::SetCommonBlock    (tag, 0,
                                     &g_TruckCommon, sizeof g_TruckCommon);

// instance setup
AsyncMobyUpdate::SetNumInstanceStreams(tag, 1);
AsyncMobyUpdate::SetOffset          (tag, 0, 0);
AsyncMobyUpdate::SetStride          (tag, 0, sizeof(TruckClass));
```

## use:

```
AsyncMobyUpdate::AddInstances      (tag, instance_block, count);
```

# add instances per frame

## setup:

```
u32 tag = AsyncMobyUpdate::AllocTag();

AsyncMobyUpdate::RegisterType      (tag, truck_frag_start, truck_frag_size);

// common setup
AsyncMobyUpdate::SetNumCommonBlocks(tag, 1);
AsyncMobyUpdate::SetCommonBlock    (tag, 0,
                                     &g_TruckCommon, sizeof g_TruckCommon);

// instance setup
AsyncMobyUpdate::SetNumInstanceStreams(tag, 1);
AsyncMobyUpdate::SetOffset          (tag, 0, 0);
AsyncMobyUpdate::SetStride          (tag, 0, sizeof(TruckClass));
```

## use:

```
AsyncMobyUpdate::AddInstances      (tag, instance_block, count);
```

# our gameplay shaders

- 32k relocatable programs
- makefile driven process combines code, data into fragment
- instance types
  - user defined (Async Moby Update)
  - predefined (Guppys, AsyncEffect)

# more shader talk

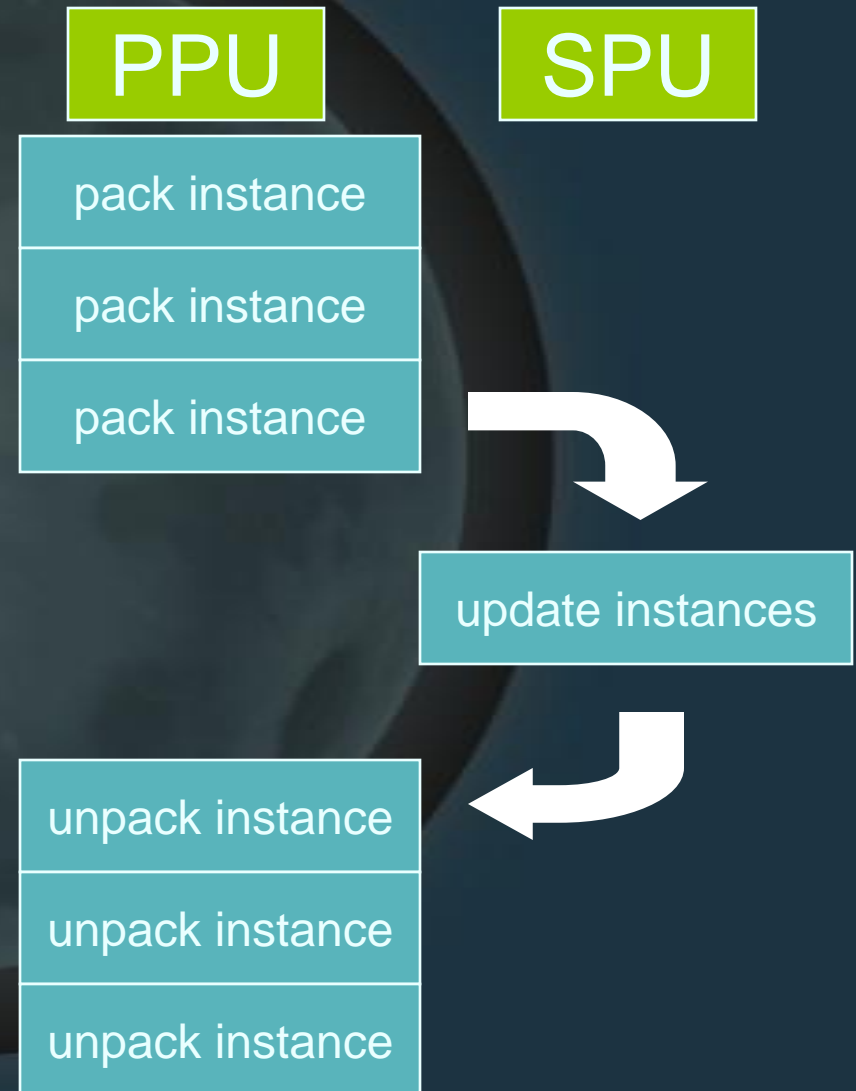
- What do our code fragments do?
  - dma up instances
  - transform instance state, position
  - maybe set some global state
  - dma down instances
- typical gameplay stuff
  - preupdate, update, postupdate

# more about instance data

- what is our instance data?
  - not an object
  - generally, a subset of an update class
  - different visibility across PU/SPU
- where does instance data live?
  - could be copied into a separate array
  - could read directly from the update classes
  - we support and use both forms

# packed instance array

- advantages
  - simplicity
  - lifetime guarantees
  - compression
- disadvantages
  - explicit fragmentation
  - pack each frame



# data inside update class

- advantages
  - pay memory cost as you go
  - don't need to know about every detail of an update class
- disadvantages
  - no longer control “lifetime” of objects
- specify interesting data with stride/offset

# instance prefetch problem

- ea\_base = starting address of our instances
- num\_instances = number of instances

```
Instance pipe[2];
dma_get(&pipe[0], ea_base, sizeof(Instance), tag);

for(int i = 0; i < num_instances; ++i) {
    Instance* cur_inst = &pipe[i&1];
    Instance* next_inst = &pipe[(i+1)&1];

    dma_sync(tag);
    dma_get(next_inst, ea_base + (i+1) * sizeof(Instance), tag);

    // ... do work
}
```

# instance prefetch problem (cont)

- ... we almost always need to fetch an associated data member out of our instances immediately

```
Instance pipe[2];
dma_get(&pipe[0], ea_base, sizeof(Instance), tag);

for(int i = 0; i < num_instances; ++i) {
    Instance* cur_inst = &pipe[i&1];
    Instance* next_inst = &pipe[(i+1)&1];

    dma_sync(tag);
    dma_get(next_inst, ea_base + (i+1) * sizeof(Instance), tag);

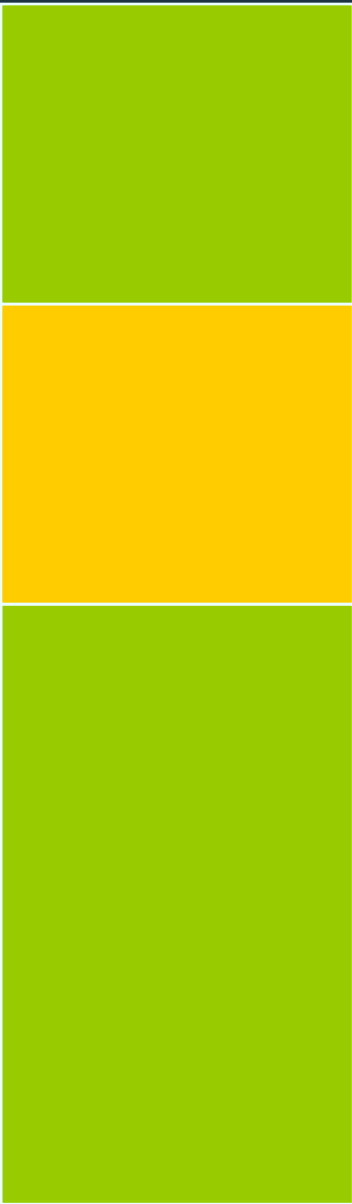
    MobyInstance cur_moby;
    dma_get(&cur_moby, cur_inst->m_moby_ea, sizeof(MobyInstance), tag);
    dma_sync(tag);

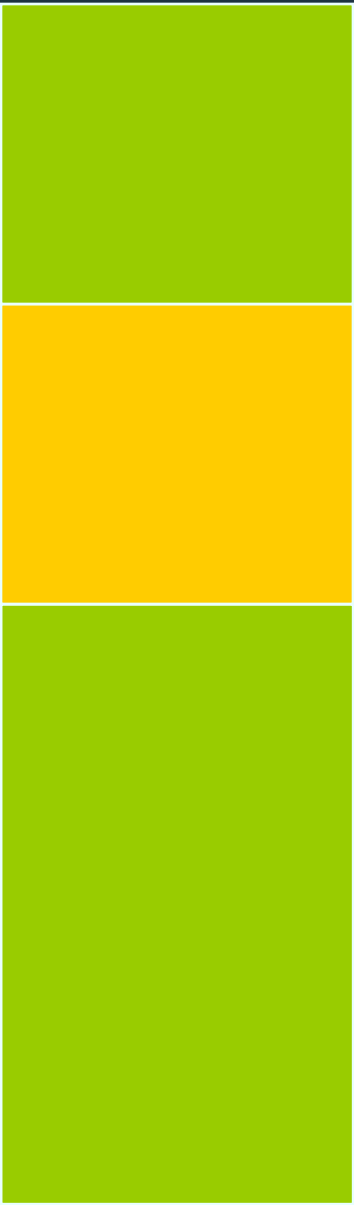
    // do work
}
```

# instance “streams”

- instances are available as “streams”
  - each has its own base, count, offset, stride, and addressing mode
- allows one to prefetch multiple associated elements without stalling
- also useful for getting slices of interesting data out of an untyped blob

# TruckUpdate (PPU) instance





**TruckInfo  
(interesting data)**





# memory addressing

- direct
  - contiguous block of instances
- direct indexed
  - indices used to deference an array of instances
- indirect indexed
  - indices used to source an array of instance pointers

# More Memory Addressing

- common blocks are preloaded
- shaders must DMA up their own instances
- meta-information is preloaded
  - indices
  - EA base pointer (direct)
  - EA pointer array (indirect)
- buffering logic is very context sensitive

# indirect indexed example

```
struct DroneInfo
{
    u32 m_stuff;
};

class DroneUpdate : public AI::Component
{
    // ...
    DroneInfo      m_info;
    MobyInstance* m_moby;
};
```

# indirect indexed example (cont)...

```
// once
AsyncMobyUpdate::SetNumInstanceStreams(amu_tag, 2);
AsyncMobyUpdate::SetStride(amu_tag, 0, sizeof DroneUpdate);
AsyncMobyUpdate::SetOffset(amu_tag, 0, OFFSETOF(DroneUpdate,
                                                m_info));
AsyncMobyUpdate::SetStride(amu_tag, 1, sizeof MobyInstance);
AsyncMobyUpdate::SetOffset(amu_tag, 1, 0);

// per frame
AsyncMobyUpdate::BeginAddInstances(amu_tag);
AsyncMobyUpdate::SetStreamIndirect(0,
    /* base */           m_updates,
    /* indices */       m_truck_update_indices,
    /* count */         m_num_trucks,
    /* max_index */     m_num_trucks);
AsyncMobyUpdate::SetStream(1, IGG::g_MobyInsts.m_array,
    m_moby_indices, m_num_trucks);
AsyncMobyUpdate::EndAddInstance ();
```

# indirect indexed example (cont)...

```
// once
AsyncMobyUpdate::SetNumInstanceStreams(amu_tag, 2);
AsyncMobyUpdate::SetStride(amu_tag, 0, sizeof DroneUpdate);
AsyncMobyUpdate::SetOffset(amu_tag, 0, OFFSETOF(DroneUpdate,
                                                m_info));
AsyncMobyUpdate::SetStride(amu_tag, 1, sizeof MobyInstance);
AsyncMobyUpdate::SetOffset(amu_tag, 1, 0);

// per frame
AsyncMobyUpdate::BeginAddInstances(amu_tag);
AsyncMobyUpdate::SetStreamIndirect(0,
    /* base */           m_updates,
    /* indices */       m_truck_update_indices,
    /* count */         m_num_trucks,
    /* max_index */     m_num_trucks);
AsyncMobyUpdate::SetStream(1, IGG::g_MobyInsts.m_array,
    m_moby_indices, m_num_trucks);
AsyncMobyUpdate::EndAddInstance ();
```

# typical usage: two streams

```
void async_foo_update(global_funcs_t* gt, update_set_info_t* info,
                    common_blocks_t* common_blocks,
                    instance_streams_t* instance_streams,
                    u8* work_buffer, u32 buf_size, u32 tags[4]) {
    u32* drone_array   = instance_streams[0].m_ea_array;
    u16* drone_indices = instance_streams[0].m_indices;
    u32  drone_offset  = instance_streams[0].m_offset;
    u32* moby_array    = instance_streams[1].m_ea_array;
    u16* moby_indices  = instance_streams[1].m_indices;
    u32  moby_offset   = instance_streams[1].m_offset;

    DroneInfo inst;
    MobyInstance moby;
    for(int i = 0; i < instance_streams[0].m_count; ++i) {
        gt->dma_get(&inst, drone_array[drone_indices[i]] + drone_offset,
                  sizeof inst, tags[0]);
        gt->dma_get(&moby, moby_array[moby_indices[i]] + moby_offset,
                  sizeof moby, tags[0]);
        gt->dma_wait(tags[0]);

        // ...
    }
}
```

1

2

# indirect indexed example #2

```
struct TruckInfo
{
    u32    m_capacity;
    u32    m_type_info;
    f32    m_hp;
    u32    m_flags;
};

class TruckUpdate : public AI::Component
{
protected:
    ...;
public:
    u32 m_ai_state;
    TruckInfo m_info __attribute__((aligned(16)));
};
```

# indirect indexed example 2 (cont)...

```
// once
AsyncMobyUpdate::SetNumInstanceStreams (amu_tag, 2);
AsyncMobyUpdate::SetStride (amu_tag, 0, sizeof TruckUpdate);
AsyncMobyUpdate::SetOffset (amu_tag, 0, OFFSETOF (TruckUpdate, m_info));
AsyncMobyUpdate::SetStride (amu_tag, 1, sizeof TruckUpdate);
AsyncMobyUpdate::SetOffset (amu_tag, 1, OFFSETOF (TruckUpdate, m_info));

// per frame
AsyncMobyUpdate::BeginAddInstances (amu_tag);
AsyncMobyUpdate::SetStreamIndirect (0,
    /* base */           m_updates,
    /* indices */        m_truck_update_indices,
    /* count */          m_num_trucks,
    /* max_index */      m_num_trucks);
AsyncMobyUpdate::SetStreamIndirect (1, m_updates,
    m_truck_update_indices,
    m_num_trucks, m_num_trucks);
AsyncMobyUpdate::EndAddInstance ();
```

# indirect indexed example 2 (cont)...

```
// once
AsyncMobyUpdate::SetNumInstanceStreams(amu_tag, 2);
AsyncMobyUpdate::SetStride(amu_tag, 0, sizeof TruckUpdate);
AsyncMobyUpdate::SetOffset(amu_tag, 0, OFFSETOF(TruckUpdate, m_info));
AsyncMobyUpdate::SetStride(amu_tag, 1, sizeof TruckUpdate);
AsyncMobyUpdate::SetOffset(amu_tag, 1, OFFSETOF(TruckUpdate, m_info));

// per frame
AsyncMobyUpdate::BeginAddInstances(amu_tag);
AsyncMobyUpdate::SetStreamIndirect(0,
    /* base */           m_updates,
    /* indices */       m_truck_update_indices,
    /* count */         m_num_trucks,
    /* max_index */     m_num_trucks);
AsyncMobyUpdate::SetStreamIndirect(1, m_updates,
    m_truck_update_indices,
    m_num_trucks, m_num_trucks);
AsyncMobyUpdate::EndAddInstance ();
```

# dma up slice of update class

```
void async_foo_update(global_funcs_t* gt, update_set_info_t* info,
                     common_blocks_t* common_blocks,
                     instance_streams_t* instance_streams,
                     u8* work_buffer, u32 buf_size, u32 tags[4])
{
1  u32* earray = instance_streams[0].m_ea_array;
  u16* indices = instance_streams[0].m_indices;
  u32  offset  = instance_streams[0].m_offset;

2  TruckInfo inst;
  for(int i = 0; i < instance_streams[0].m_count; ++i)
  {
    gt->dma_get(&inst, earray[ indices[i] ] + offset,
               sizeof inst, tags[0]);
    gt->dma_wait(tags[0]);

    // update
  }
}
```

# dma full update class, slice info out

```
u32* earray = instance_streams[0].m_ea_array;
u16* indices = instance_streams[0].m_indices;
u32  offset = instance_streams[0].m_offset;
u32  stride = instance_streams[0].m_stride;
```

1

```
u32* ai_earray = instance_streams[1].m_ea_array;
u16* ai_indices = instance_streams[1].m_indices;
u32  ai_offset = instance_streams[1].m_offset;
```

2

```
u8* blob = (u8*) alloc(instance_streams[1].m_stride);
for(int i = 0; i < instance_streams[0].m_count; ++i)
{
    gt->dma_get(&blob, earray[ indices[i] ], stride, tags[0]);
    gt->dma_wait(tags[0]);
}
```

3

```
TruckInfo *inst = (TruckInfo*) (blob + instance_streams[0].m_offset);
u32 *ai_state = (u32*) (blob + instance_streams[1].m_offset);
```

```
// ...
}
```

# code fragment signature

```
extern "C" void
async_foo_update(global_funcs_t* gt,
                 update_set_info_t* info,
                 common_blocks_t* common_blocks,
                 instance_streams_t* instance_streams,
                 u8* work_buffer, u32 buf_size, u32 tags[4])
```

- global\_funcs\_t - global function pointer table
- update\_set\_info\_t - meta info
- common\_blocks\_t - stream array for common blocks
- instance\_streams\_t - stream array for instances
- work\_buffer & buf\_size - access to LS
- dma\_tags - 4 preallocated dma tags

# guppys

- lightweight alternative to MobyInstance
- update runs entirely on SPU
- one 128byte instance type
- common data contained in “schools”
- simplified rendering

# guppys



- common use case: “bangles”
- to cleave a mesh, we previously required an entire new MobyInstance to cleave a mesh
  - turn off arm mesh segment on main instance
  - turn off all other mesh segments on spawned instance
- spawn a guppy now instead

# the guppy instance

- position/orientation EA
- 1 word flags
- block of “misc” float/int union data
- animation joints EA
- joint remap table
- Insomniac “special sauce”

# Async Effect

- simplified API for launching SPU-updating effects
- no code fragment writing necessary
- specialized at initialization
  - linear/angular movement
  - rigid body physics
  - rendering parameters

# Async Effect API

```
u16 moby_iclass = LookupMobyClassIndex( ART_CLASS_BOT_HYBRID_2_ORGANS );
MobyClass* mclass = &IGG::g_MobyCon.m_classes[ moby_iclass ];

u32 slot = AsyncEffect::BeginEffectInstance( AsyncEffect::EFFECT_STATIONARY,
                                             mclass );
AsyncEffect::SetEffectInstancePo ( slot, moby->m_mat3, moby->m_pos );
AsyncEffect::SetEffectInstanceF32( slot, AsyncEffect::EFFECT_PARAM_LIFESPAN,
                                   20.0f );
u32 name = AsyncEffect::Spawn( slot );
```

- stationary effect with 20 second life
- **name** can be used to kill the effect

# SPU invoked code



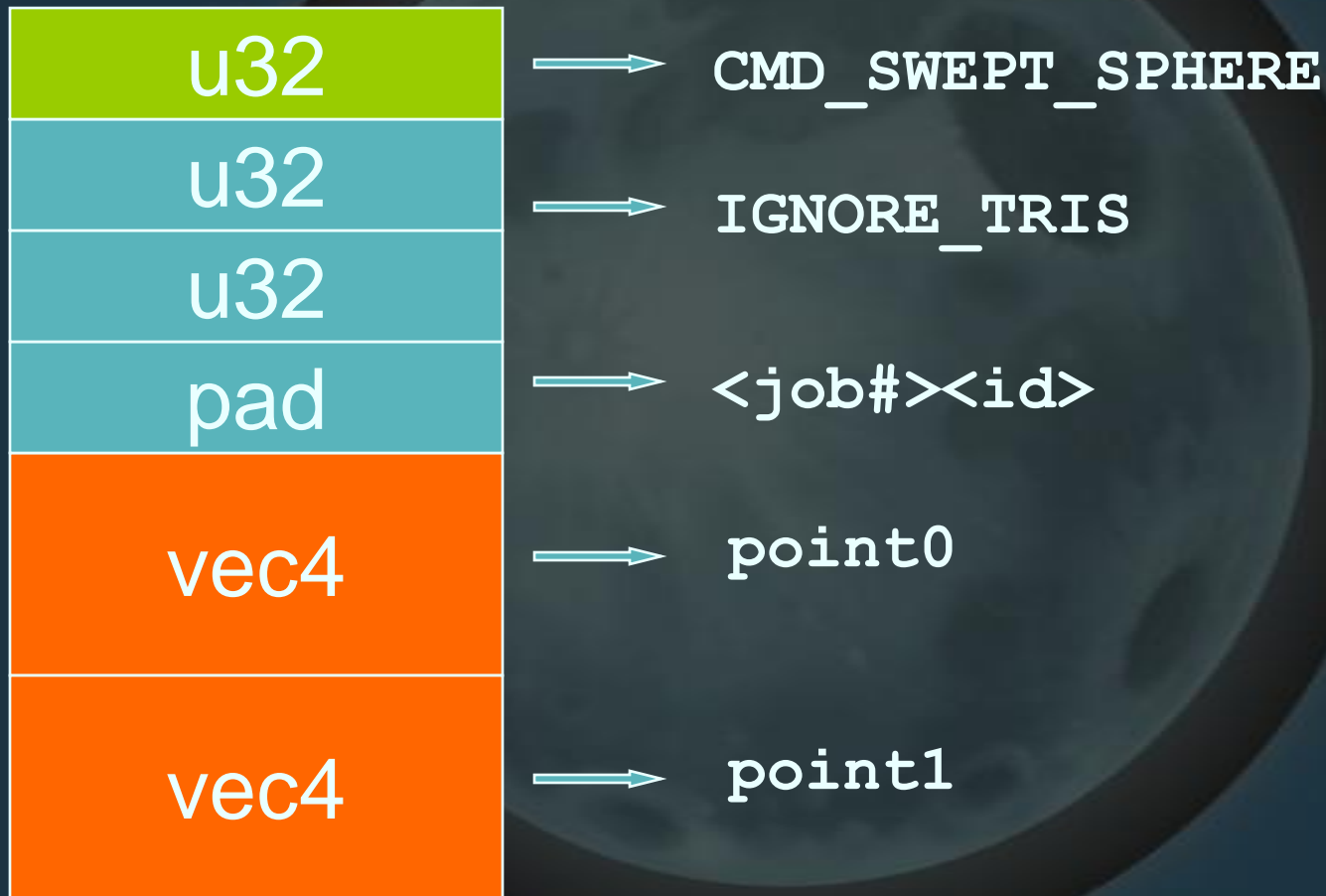
# different mechanisms

- immediate
  - via global function table
- deferred
  - command buffer
- adhoc
  - PPU shims
  - direct data injection

# deferred

- PPU shims
  - flags set in SPU update, queries/events triggered subsequently on PPU
- command buffer
  - small buffer in LS filled with command specific byte-code, flushed to PPU
- atomic allocators
  - relevant data structures packed on SPU, atomically inserted

# command buffer: swept sphere



# command buffer: results

frame n

```
handle_base = ((stage << 5) | (job_number & 0x1f)) << 24;  
// ...  
handle = handle_base + request++;
```

frame n+1

offset table

frame n, stage 0, job 1

frame n, stage 1, job 1

frame n, stage 2, job 1

frame n, stage 0, job 2

result



# direct data

- patch into atomically allocated, double buffered data structures
- instance allocates fresh instance each frame, forwards state between old and new
- deallocation == stop code fragment
- used for rigid body physics

# direct data

SPU

main memory

1

```
new_ea = rigid body #0
```

2

```
get(&ls_rigidbody, old_ea)
```

```
update ls_rigidbody
```

```
put(&ls_rigidbody, new_ea)
```

3

```
old_ea = new_ea
```

rigid body #0

rigid body #1

rigid body #2

unallocated



# SPU API

- SPU-API
  - mechanism to expose new functionality through to the AsyncMobyUpdate system
  - library of code fragment code and common data (“fixup”)
  - function pointer table (“interface”) oriented
  - hides immediate or deferred commands

# SPU API

h

```
struct rcf_api_interface
{
    u32 (*derived_from)(u32 update_class, u32 base_class);
    u32 (*add_bolts)    (u32 update_class, u32 value);
};
```

cpp

```
rcf_api_interface* rcf_api = gt->get_spu_api("rcf2");

if(rcf_api->derived_from(inst->m_update_class, HERO_Ratchet_CLASS))
{
    rci_api->add_bolts(inst->m_update_class, 25);
}
```

# example #1

- Jets
  - uses AsyncMobyUpdate along with an Update Class
  - packed instance array
  - code fragment per AI state
  - little initialization setup, state changes rare
  - events triggered using adhoc method
    - flags checked at pack/unpack time
- inputs:
  - position/orientation
  - state info
- output:
  - position/orientation

# example #2

- zombie limbs
  - guppy
  - not much update logic
  - direct data interface to rigid body physics
- input:
  - position/orientation
  - animation joints
  - collision info
- output:
  - packed rigid bodies

porting code



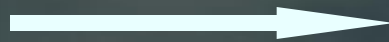
# porting code sucks

- difficult to retrofit code
- different set of constraints
- expensive use of time
- can result in over-abstracted systems
  - like, software cache for transparent pointers

# couple of tips

Separate interesting information from non-necessary data structures

```
struct foo_t : bar_t
{
    // stuff
    u32 m_first;
    u32 m_second;
    u32 m_third;
    u32 m_fourth;
    // more stuff
};
```



```
struct foo_info_t
{
    u32 m_first;
    u32 m_second;
    u32 m_third;
    u32 m_fourth;
};

struct foo_t : bar_t
{
    // stuff
    foo_info_t m_info;
    // more stuff
};
```

# couple of tips (cont)...

avoid pointer fixup, define pointer types to unsigned ints

```
struct baz_t
{
    u32          m_foo;
#ifdef PPU
    MobyInstance* m_moby;
#else
    u32          m_moby_ea;
#endif
    f32          m_scale;
};
```

# living with polymorphism

- the described mechanisms have problems with virtual functions
  - would need to port and patch up all possible vtable destinations
  - would end up duplicating/shadowing virtual class hierarchy on SPU
- could work, but we don't do that

# compile-time polymorphism

- do a trace of the class hierarchy: one code fragment per leaf class
- separate base functions into .inc files
- virtual functions selected through sequential macro define/undef pairs
- not described:
  - deferred resolution of base function calls to derived function via preprocessor pass

# living with polymorphism (cont)

## pickupbolt\_preupdate.cpp

```
#include "code_fragment_pickup.inl"  
#include "code_fragment_pickup_bolt.inl"
```

## code\_fragment\_pickup.inl

```
void base_on_pickup(CommonInfo* common, InstanceInfo* inst) {  
    inst->m_base_info->m_spu_flags |= PICKUP_SPU_FLAGS_PICKED_UP;  
}  
  
#define ON_PICKUP(c,i)    base_on_pickup(c, i)
```

## code\_fragment\_pickup\_bolt.inl

```
void bolt_on_pickup(CommonInfo* common, InstanceInfo* inst) {  
    common->m_active_bolt_delta--;  
}  
#undef ON_PICKUP  
#define ON_PICKUP(c,i)    bolt_on_pickup(c, i)
```

# living with polymorphism (cont)

## pickupbolt\_preupdate.cpp

```
#include "code_fragment_pickup.inl"  
#include "code_fragment_pickup_bolt.inl"
```

## code\_fragment\_pickup.inl

```
void base_on_pickup(CommonInfo* common, InstanceInfo* inst) {  
    inst->m_base_info->m_spu_flags |= PICKUP_SPU_FLAGS_PICKED_UP;  
}  
  
#define ON_PICKUP(c,i)    base_on_pickup(c, i)
```

## code\_fragment\_pickup\_bolt.inl

```
void bolt_on_pickup(CommonInfo* common, InstanceInfo* inst) {  
    common->m_active_bolt_delta--;  
}  
#undef ON_PICKUP  
#define ON_PICKUP(c,i)    bolt_on_pickup(c, i)
```

# living with polymorphism (cont)

## pickupbolt\_preupdate.cpp

```
#include "code_fragment_pickup.inl"  
#include "code_fragment_pickup_bolt.inl"
```

## code\_fragment\_pickup.inl

```
void base_on_pickup(CommonInfo* common, InstanceInfo* inst) {  
    inst->m_base_info->m_spu_flags |= PICKUP_SPU_FLAGS_PICKED_UP;  
}  
  
#define ON_PICKUP(c,i)    base_on_pickup(c, i)
```

## code\_fragment\_pickup\_bolt.inl

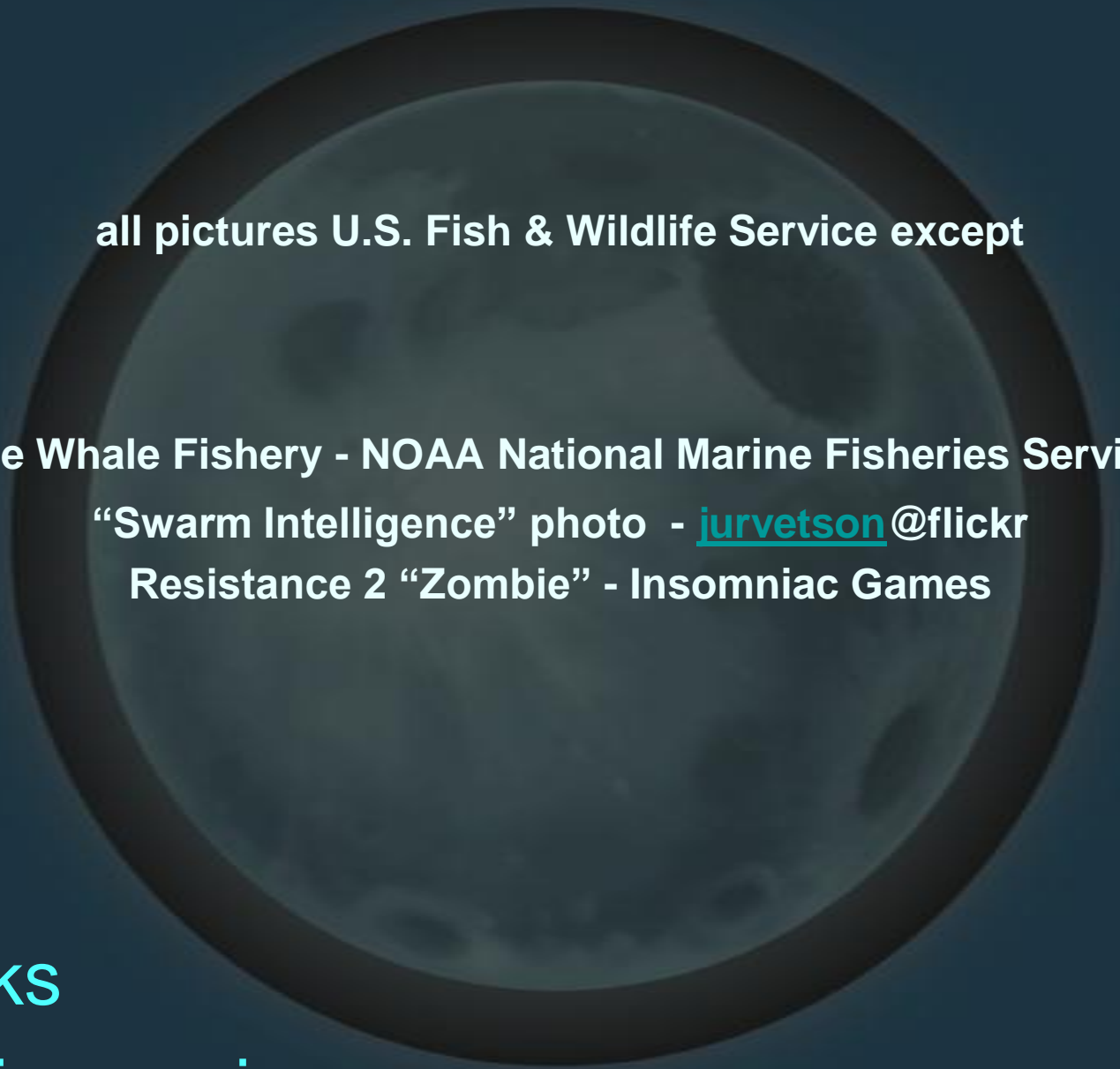
```
void bolt_on_pickup(CommonInfo* common, InstanceInfo* inst) {  
    common->m_active_bolt_delta--;  
}  
#undef ON_PICKUP  
#define ON_PICKUP(c,i)    bolt_on_pickup(c, i)
```

# design from scratch

- parameterized systems
  - not specialized by code
- use atomic allocators as programming interfaces
- no virtual functions in update phase
- separate perception, cognition, action
- plan to interleave ppu/spu

# In conclusion

- design up front for deferred, SPU-friendly systems.
- don't worry too much about writing optimized code, just make it difficult to write unoptimizable code
- remember, this is supposed to be fun. SPU programming is fun.



all pictures U.S. Fish & Wildlife Service except

The Whale Fishery - NOAA National Marine Fisheries Service

“Swarm Intelligence” photo - [jurvetson](#)@flickr

Resistance 2 “Zombie” - Insomniac Games

Thanks

[joe@insomniacgames.com](mailto:joe@insomniacgames.com)