

Async [CharacterX]

Implementing the [CharacterX]
using AsyncMobyUpdate

Daniel Gonzalez

2/29/08

Conclusion:

- Need to look at old problems in new ways.
 - Turn off the autopilot.
- Only as complex as you make it.
 - Remember the goal is simply to move more code onto the SPUs.
 - Chip away at the problem rather than re-architecting whole systems.

Code fragment / SPU shader

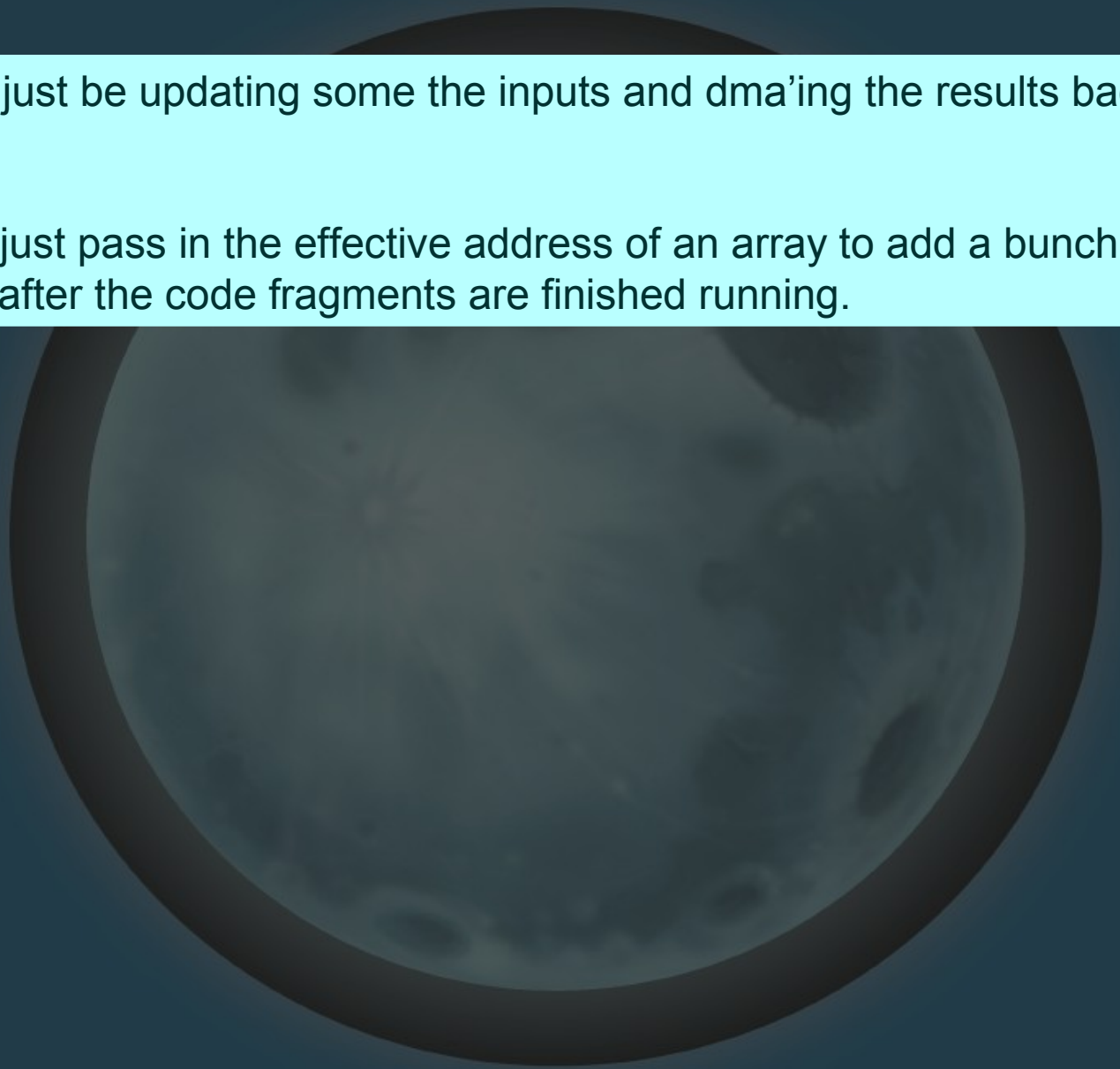
- Transforms a set of inputs into outputs.
- Input is generally an array of packed instance data.
 - Only what you need, not entire update class instances.
 - Velocity, acceleration, and position of a set of vehicles.
 - State and vitals of a set of bots.

A code fragment is basically just a function that gets passed the location of your input data in main memory. Not calling out to other systems. Just transforming the data.

Packing != compressing. Not doing anything particularly clever to the data, you shouldn't be spending time on the ppu preparing the data for the spu. On the ppu you're just copying what's needed into an array, and in the code fragment you're dma'ing that data into the local store.

Code fragment / SPU shader

- Outputs –
 - Might just update the inputs.
 - New position, orientation, velocity.
 - Not necessarily a mirror of inputs though.
 - Batch of collision requests or animation queries.
 - State changes.



You might just be updating some the inputs and dma'ing the results back into main memory.

You could just pass in the effective address of an array to add a bunch of queries to be sent after the code fragments are finished running.

Code fragment / SPU shader

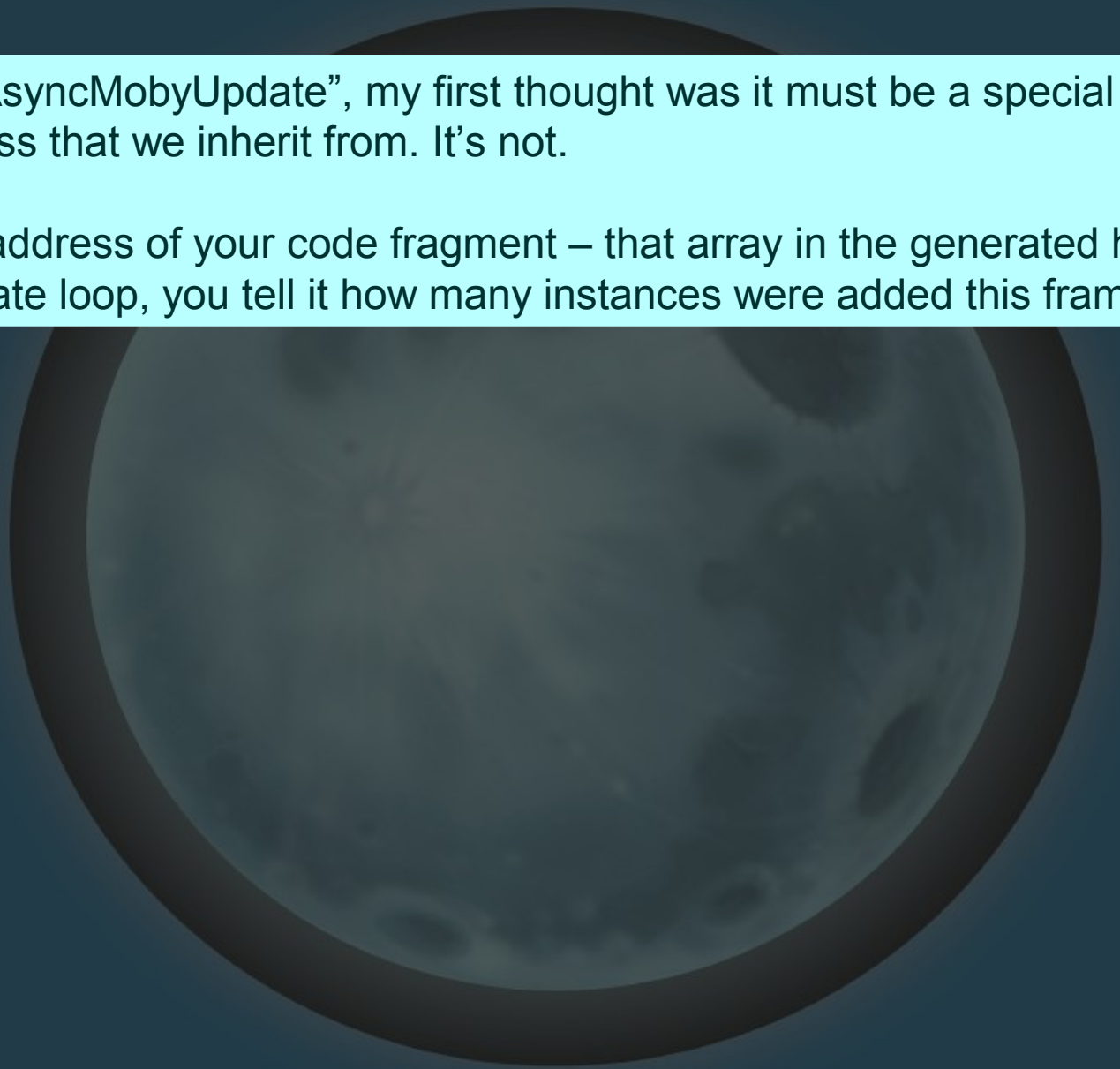
- Code fragment itself is just a piece of data.
 - Header generated with an array containing machine code of the compiled code fragment.
 - Dma'd to the local store of an SPU by the AsyncMobyUpdate system when its time to run the code fragment.

Important to keep this in mind because the code, the stack, and the input data you're dma'ing in has to fit into the spu's local store. If you have a massive code fragment, it'll limit the number of instances you can load in and iterate over at a time.

When the `code_fragments` project builds, it will generate a header defining an array that contains the machine code of the code fragment.

What is AsyncMobyUpdate?

- It's not a GameMobyUpdate.
 - Your update class doesn't inherit from it.
- It's a service that handles the dirty work of running code fragments.
 - Tell it the address of your code fragment and instance data.
 - Add instances during job collection phase of update loop.



Hearing “AsyncMobyUpdate”, my first thought was it must be a special moby update class that we inherit from. It’s not.

Tell it the address of your code fragment – that array in the generated header. In the update loop, you tell it how many instances were added this frame.

AsyncMobyUpdate

- After collecting job requests, it kicks them off on an SPU using voodoo / ancient Chinese secret.
 - Dma's in your code.
 - Pushes the instance count and the address of your instance data onto the stack.
 - Sets the instruction pointer to the start of your code.

Retrofitting code fragments into existing code base

- Typical gameplay code –
 - Talks to other systems whenever it pleases.
 - Gets immediate results.
- Code fragment –
 - Exists in a vacuum while running.
 - Its only communication is through its inputs and outputs.

The main problem.

Talks to other systems, gets immediate results. It's actually kind of hard to find a decent sized block of code that isn't like this, and is just self contained.

The problem is, this type of code doesn't translate easily into code fragments.

Can dma in and out DATA, but can't communicate with other systems because there is no other code resident in the local store other than some AsyncMobyUpdate system code, and some helper functions like debug printing.

Retrofitting code fragments

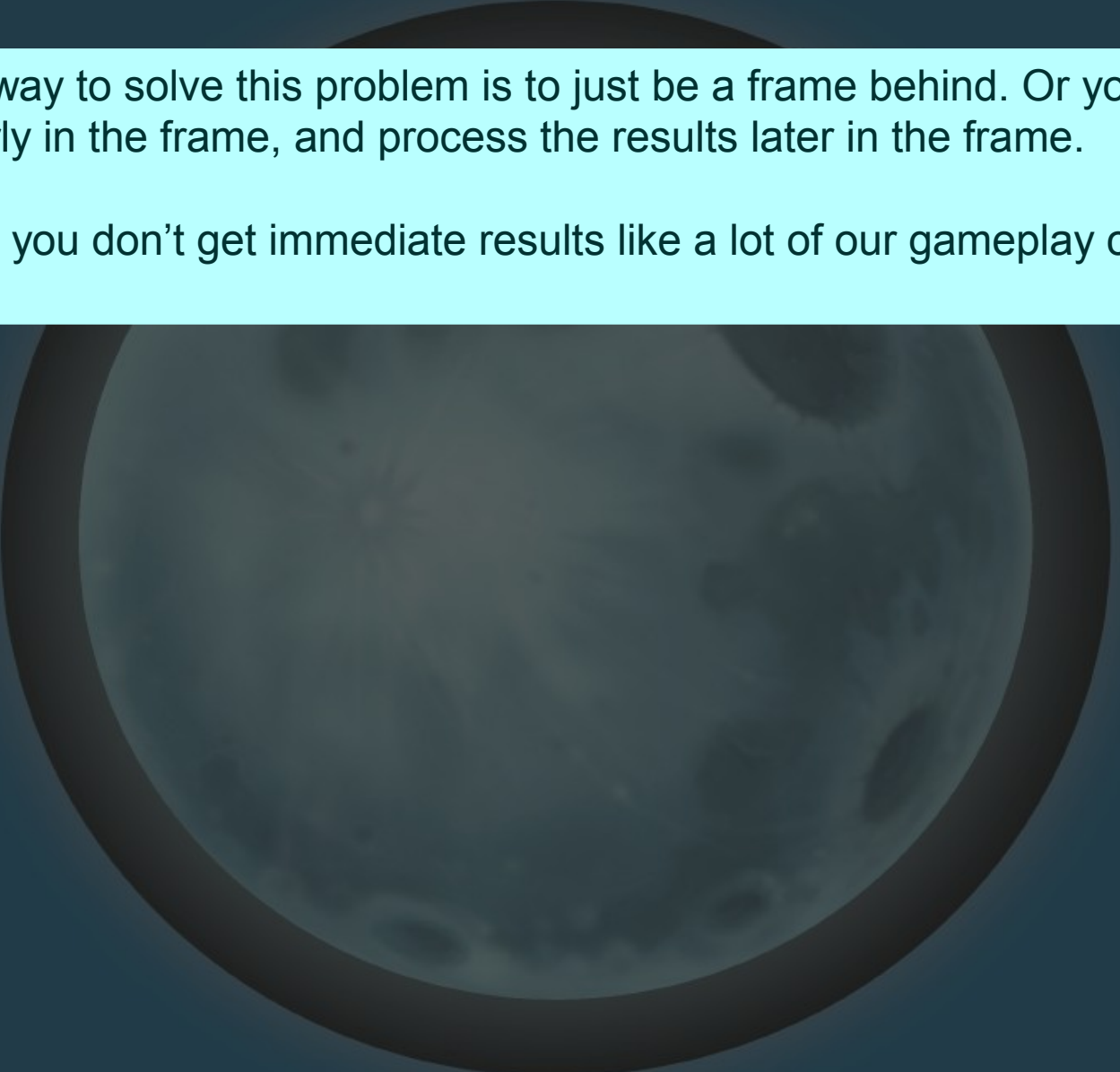
- Code fragment –
 - Won't receive any query results until next update.
 - Any PPU-side code waiting on results from the code fragment will have to wait until `AsyncMobyUpdate::Sync`.

So if you do communicate through your outputs, you won't receive the results of those queries until the next frame.

Need to wait until the code fragment jobs are done before you can use the results on the ppu.

The usual solution

- Use last frame's result.
 - In a lot of cases, this is acceptable.
 - Targeting aim prediction a frame behind.
 - Movement physics a frame behind.

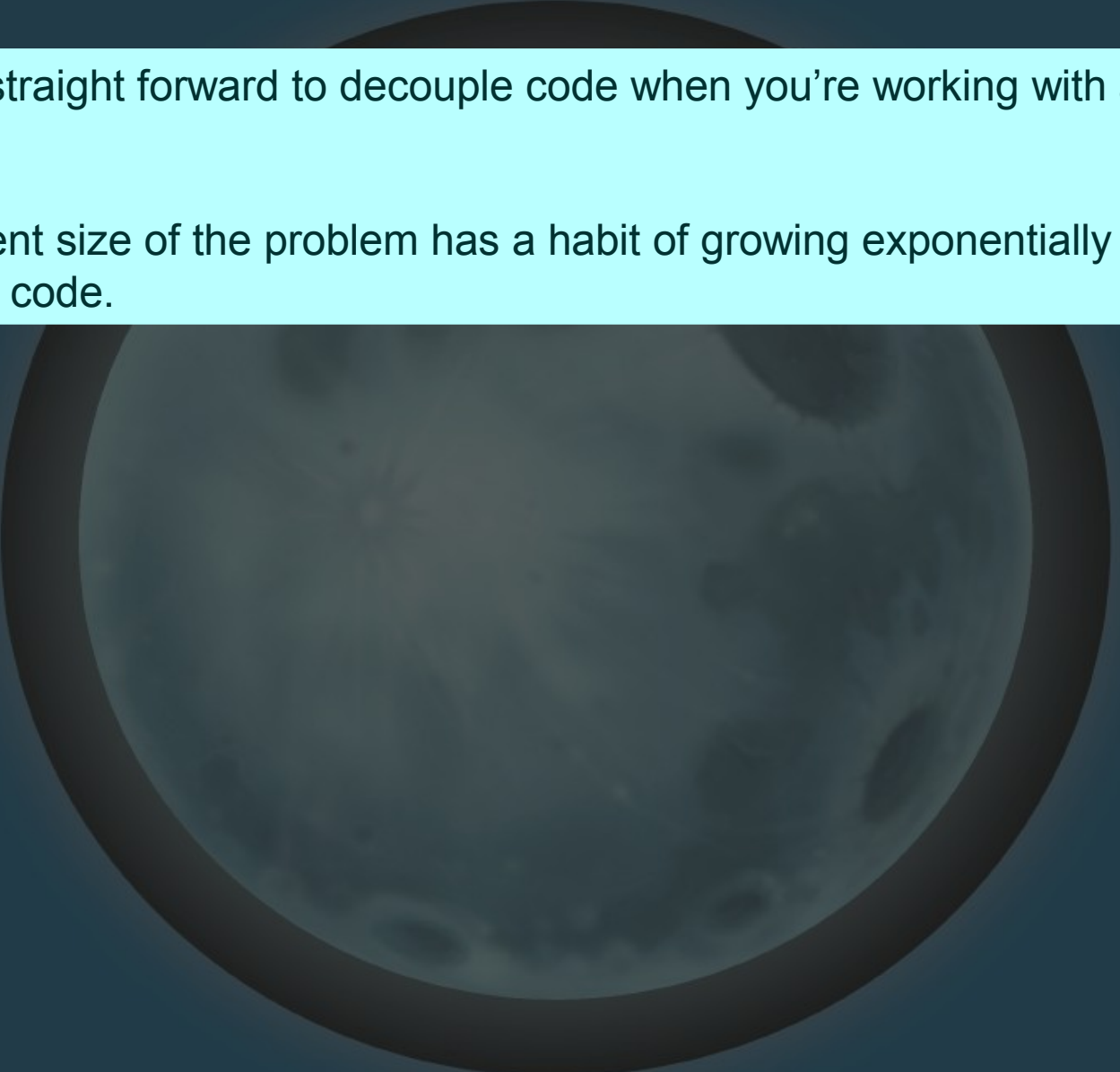


The usual way to solve this problem is to just be a frame behind. Or you can kick off jobs early in the frame, and process the results later in the frame.

Either way, you don't get immediate results like a lot of our gameplay code relies on.

Great! Ok, let's do that

- It's not always so straight forward.
 - That function you went to rewrite turned out to call three other functions. Including a virtual function call.
 - Those three functions are probably coupled to other systems through yet more function calls.
 - So keep digging until you find a leaf.
 - » How deep are you at this point?
 - » What system are you in?
 - Spent a lot of time doing this type of analysis on [X_BaseClass] and its various sub-systems.



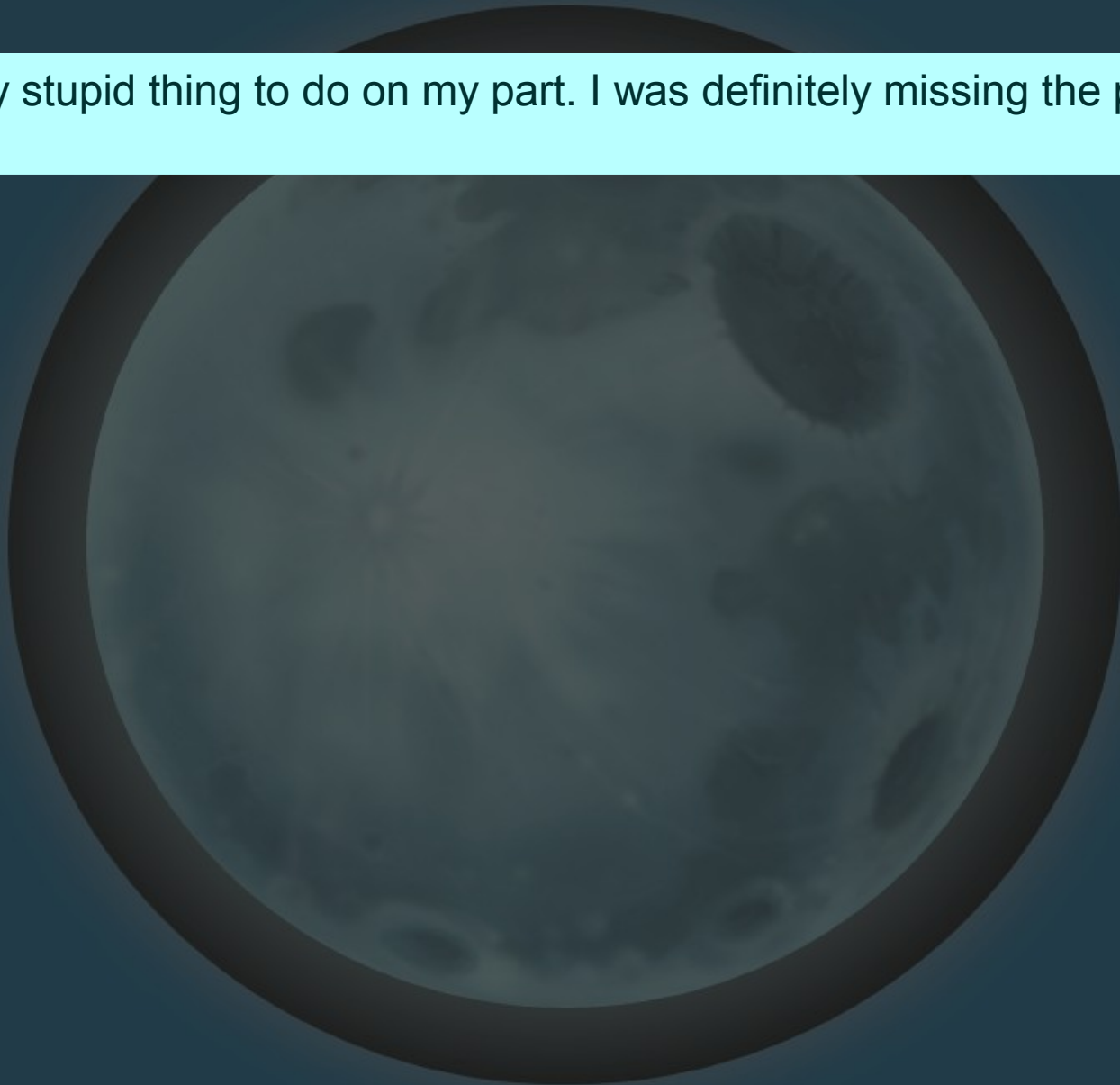
It's not so straight forward to decouple code when you're working with a large code base.

The apparent size of the problem has a habit of growing exponentially the more you dig into the code.

That was non, non non, non heinous

- The goal at this point is simply to get **MORE** gameplay running on the SPUs.
 - Not necessarily ALL the code of an update class.
 - Don't try to redesign large pieces of code in one big push.
 - If you find yourself digging deep into multiple sub-systems, stop yourself before you lose your mind.

Completely stupid thing to do on my part. I was definitely missing the point for a while.



Chipping away

- Look for pieces of code that are already a good fit for moving to a code fragment.
 - Small well defined problems.
 - Few dependencies. All inputs known at start of execution.
- If you are going to make entire update classes out of code fragments, should be simple objects with simple needs.

Examples

- [CharacterX] idle motion calculation.
 - Simple data transform.
 - Inputs –
 - Current phase and offset.
 - Frequency.
 - Outputs –
 - Updated phase and offset.

Examples

- [CharacterX] squirrely [projectile] physics.
 - Again, straight forward data transform.
 - Inputs –
 - Position and orientation.
 - Velocity.
 - Target position.
 - Steer direction.
 - Squirrelliness.
 - Etc...

Squirrely [projectile] physics

- [CharacterX] squirrely [projectile] physics.
 - Outputs –
 - Updated position and orientation.
 - Updated steer direction.
 - Updated velocity.

Using code fragments in practice

- Necessary steps -
 - PPU side -
 - Packing and unpacking on the ppu.
 - Adding jobs/instances.
 - SPU side –
 - Write the code fragment!

GameplayFragment

- Simplifies the process of creating and using a code fragment.
- Decouples update classes from AsyncMobyUpdate.
 - Makes it more straight forward to add code fragments to existing update classes without changing the update class to support this.

Using GameplayFragment: PPU – DestFromPath.h

```
#include "gameplay_fragment/GameplayFragmentSystem.h"

class DestFromPath : public GameplayFragment
{
    DECLARE_GAMEPLAY_FRAGMENT(DestFromPath, PreUpdate);
public:
    void AddInstance( CharacterYUpdate* character_y );
};
```

Example from the [Character Y] for creating a code fragment that sets a destination based on a path. There's another include or two, but this is basically it.

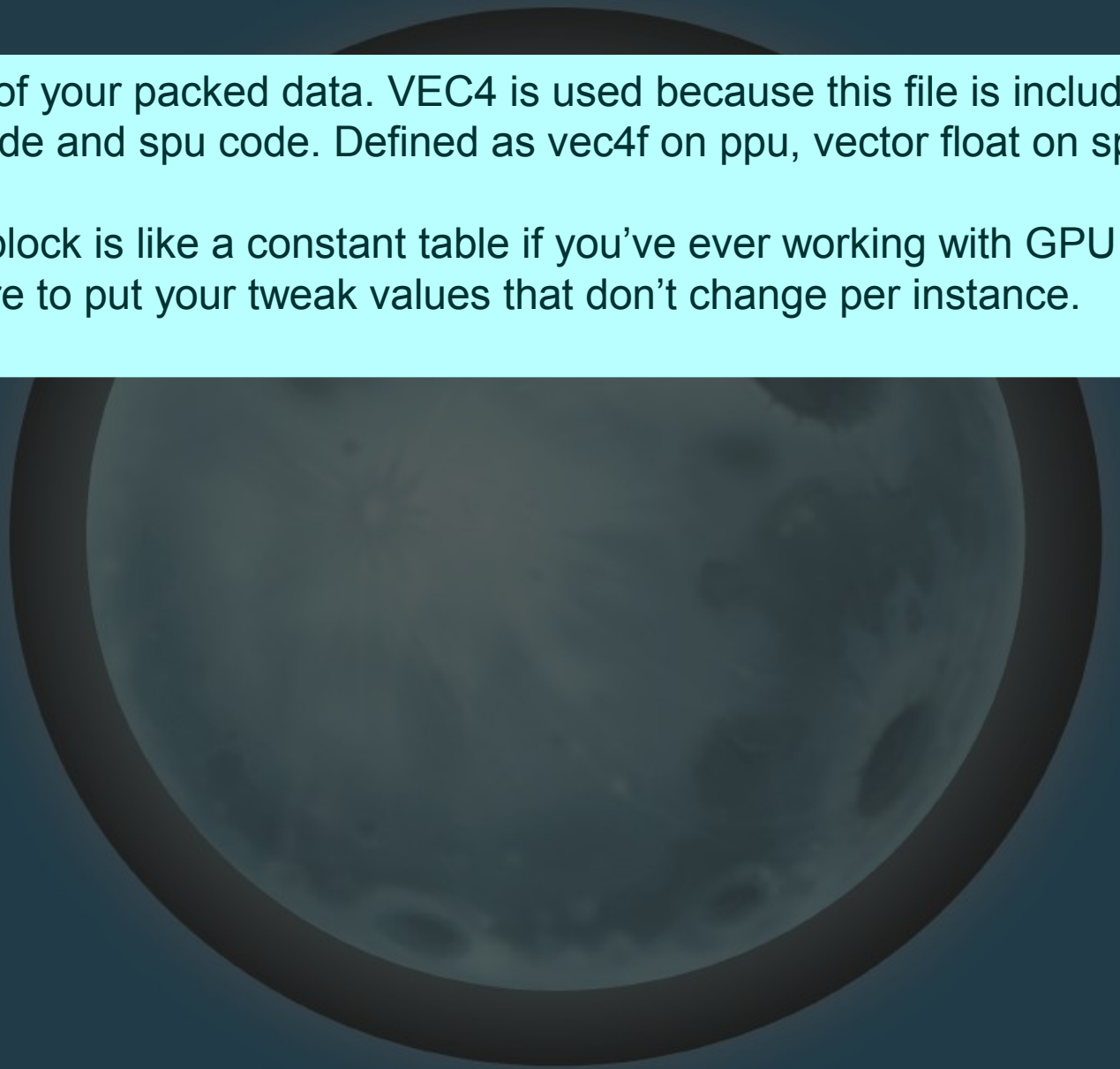
Cheesy macro to do some evil stuff that you don't need to worry about. Tell it which phase you're going to use it (support for Update phase in the works), and when the AsyncMobyUpdate system is collecting jobs in the update loop, this will decide which phase to collect jobs for this code fragment.

AddInstance is the interface through which your update class adds instances for the code fragment to process. You define its signature – this is the data it's going to pack.

Using GameplayFragment: PPU/SPU – CharacterYShared.h

```
struct DestFromPathData
{
    VEC4 m_part_pos;
    u32  m_path_ea;
    u32  m_path_count;
    u32  m_current_node;
    u32  m_dest_ea;
};

struct DestFromPathCommon
{
    f32 m_at_dest_tol;
    f32 m_pad[3];
};
```



Signature of your packed data. VEC4 is used because this file is included in both the ppu code and spu code. Defined as `vec4f` on ppu, `vector float` on spu.

Common block is like a constant table if you've ever working with GPU shaders. Somewhere to put your tweak values that don't change per instance.

Using GameplayFragment: PPU - DestFromPath.cpp

```
DEFINE_GAMEPLAY_FRAGMENT(DestFromPath);

static DestFromPathData    s_data[MAX_INSTANCES];
static DestFromPathCommon s_common_block;
static CharacterYUpdate*   s_character_ys[MAX_INSTANCES];
static u32                  s_num_fragment_instances;

void DestFromPath::Init()
{
    INITIALIZE_GAMEPLAY_FRAGMENT(DestFromPath);
    RegisterFragment( (u32)async_dest_from_path, s_data,
                      &s_num_fragment_instances, s_common_block );
    SetSortKey( 0 );
}
```

More evil macro stuff. Static data – this is the array of instances you'll be dma'ing into your code fragment.

RegisterFragment is basically a wrapper for registering the fragment with the AsyncMobyUpdate system, plus other GameplayFragment glue code.

SetSortKey is used to set the order that code fragments are run in by the AsyncMobyUpdate system. In this case I want this fragment to update the character_ys destination before the MoveToDestination fragment runs, so in the MoveToDestination fragment I set the sort key to 1. This is why I pass the effective address of the character_ys destination rather than packing it, so the MoveToDestination fragment can read the result.

Packing the data – PPU – DestFromPath.cpp cont.

```
void DestFromPath::AddInstance( CharacterYUpdate* character_y )
{
    s_character_ys[s_num_instances] = character_y;

    s_data[s_num_instances].m_current_node = character_y->m_current_node;
    s_data[s_num_instances].m_part_pos = character_y->m_moby->m_pos;

    Path* path = PathGetPtr( character_y->m_path_handle );
    s_data[s_num_instances].m_path_count = path->m_point_count;
    s_data[s_num_instances].m_path_ea = (u32)path->m_points;
    s_data[s_num_instances].m_dest_ea = (u32)&character_y->m_move_dest;
    ++s_num_instances;
}
```

No macros here!

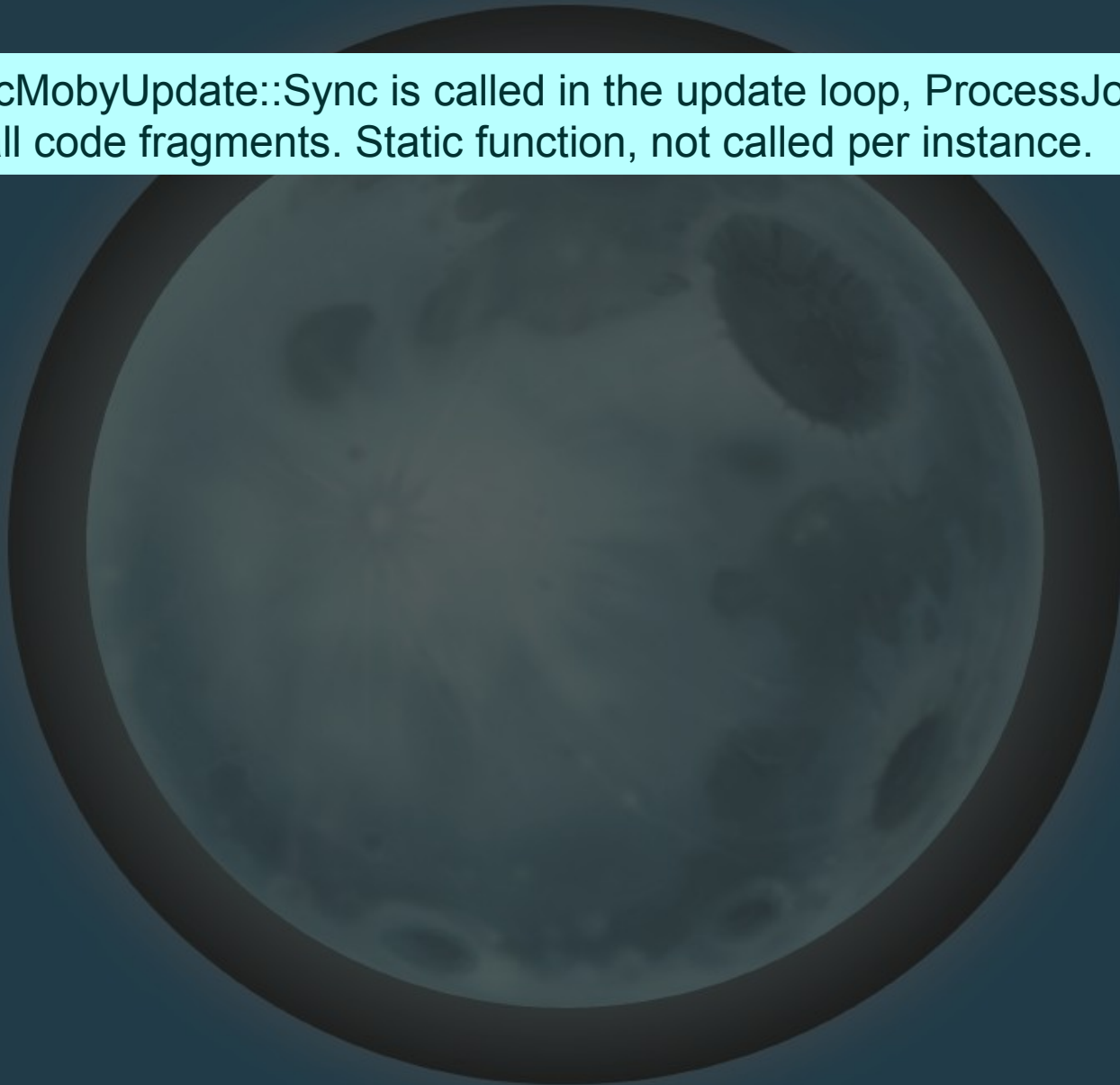


Processing the results – PPU – DestFromPath.cpp cont.

```
void DestFromPath::ProcessJobResults()
{
    for( u32 i = 0; i < s_num_instances; ++i )
    {
        s_character_ys[i]->m_current_path_node = s_data[i].m_current_node;
    }

    s_num_instances = 0;
}
```

After `AsyncMobyUpdate::Sync` is called in the update loop, `ProcessJobResults` is called on all code fragments. Static function, not called per instance.



Writing the code fragment SPU – dest_from_path.cpp

```
extern "C"  
void async_dest_from_path(global_funcs_t* gt, u32 tag, u8* common_block,  
                          u32 ea, u32 count, u8* work_buffer, u32 buf_size,  
                          u32 dma_tags[4] )  
{  
    u32 block_size = buf_size / 32;  
    DestFromPathData* instance_data = (DestFromPathData*)work_buffer;  
    DestFromPathCommon* common = (DestFromPathCommon*)common_block;  
    while( count )  
    {  
        u32 num_instances = MIN( block_size, count );  
        gt->dma_get( instance_data, ea, num_instances * 32, dma_tags[0] );  
        gt->dma_sync( dma_tags[0] );  
        for( u32 i = 0; i < num_instances; ++i )  
            // do stuff  
        gt->dma_put( instance_data, ea, num_instances * 32, dma_tags[0] );  
        count -= num_instances;  
        ea    += num_instances * 32;  
    }  
}
```

- Gt is a global function table. Includes the helper functions to do dma's, and for debug printing.
- Tag is something new, ask Joe why that's there.
- Ea is the effective address of your instance data array in main memory.
- Count is the number of instances in that array .
- Work buffer is a piece of memory in the local store that you can use for dma'ing in your data.
- Buf_size is how big that buffer is.
- Dma_tags – I don't know if they serve some other purpose, but in practice they're synchronization primitives you can use for your dma's.

This is what the typical code fragment is going to look like stripped of its functionality. It's the code necessary to read in the instance data.

Block_size is the largest block of instances you can dma into the work buffer at a time. Casting work buffer to the packed data type because this is where I'm going to dma in the instance data array.

Writing the code fragment – SPU – dest_from_path.cpp cont.

```
DestFromPathData* character_y_data = &instance_data[i];
vector float dist = character_y_data->m_part_pos - current_node_pos;
dist = spu_dot3(dist,dist);
dist = spu_shuffle(dist, dist, shuffle_X0X0X0X0 );
vector unsigned int mask = spu_cmpgt(dist,
                                     spu_splats(common->m_at_dest_tol));
vector float new_dest = spu_sel( next_node, current_node, mask );

gt->dma_put( &new_dest, character_y_data->m_dest_ea, 16, dma_tags[3] );
```

Using the code fragment – PPU – CharacterY.cpp

```
#include "update/etc/DestFromPath.h"

void CharacterYUpdate::Init()
{
    m_dest_from_path_fragment = DestFromPath::Alloc();
}

void CharacterYUpdate::PreUpdate()
{
    m_dest_from_path_fragment->AddInstance(this);
}
```



That's all I've got!
Questions?