

Introducing SPU Shaders

Mike Acton, Insomniac Games

State of affairs:

- Most major engine systems are already on the SPU.
- SPU code and data optimization is fairly well understood at this point.
- While not everything has been optimized, we do know what needs to be done.

In moving to the second generation of Playstation 3 games and beyond, we have a new set of challenges:

- Overall reduction of synchronization points between the SPUs and the PPU and RSX.
- Simple methods for encouraging the use of the SPUs by more, or most, systems.
- Keeping SPU code and data design straightforward and fast.
- Make forward progress toward *all* code running as asynchronously as possible.

What are SPU Shaders?

Specifically, an **SPU Shader is a fragment of SPU code that fits into a larger system to do something specific or custom**. The system itself may use a series of SPU shaders to do the work, and/or users of the system may inject SPU Shaders to do custom processing.

In the simplest example, SPU Shaders can be thought of as a replacement for callbacks. General callbacks (i.e. to PPU code) create extra, unnecessary synchronization between the PPU and SPUs. Injecting SPU code into the system relieves the burden.

Typical SPU System.
(Stream processing a few different types of data.)



1. Need a small change to one of the instances of data.
2. Make an exception in the system.
3. The system grows.



Or instead,

1. Create a small code fragment ("Shader") that handles the exception.
2. Load up the code with the data.
3. The system is not affected.



What *aren't* SPU Shaders?

- SPU Shaders are not a generic solution.
- SPU Shaders are not a big system or library of code.

SPU Shaders are more of a policy, or an idea. A very basic idea where we believe that it is exactly the generic solutions and big monolithic systems that “solve all your problems for you” that cause most of the trouble. The idea is to go beyond “keep it simple” and to “simple as design policy.”

The interface to an SPU Shader (i.e. the “main” function’s parameters) are not generic. They are specific to a system or sub-system – Each system accepts shaders that match the requirements specific to it. E.g.

1. The parameter list
2. The maximum code size of a shader
3. The maximum scratch memory available to a shader

4. Whether or not there's any global state associated with shader calls (likely not though)

However, as policy, some things are constant to keep things as simple as possible:

1. Per system or sub-system, the parameter list for the shaders is fixed (i.e. no dynamic parameter lists depending on data value)
2. The working buffer for a shader is fixed per sub-system (i.e. shaders can depend on having a specific amount of memory, but how much that is, depends on the sub-system.)
3. Control is always in the shader-writer's hands. i.e. Shader systems do not try to implement "helpers" that take away control from the shader writer ("don't repeat the mistakes of SPURS").
 1. E.g. if shaders need dma tags, then dma tags are passed into the shaders as parameters – rather than a kind of "GetFreeDmaTagFromParentSystem()" system.
 2. E.g. Systems do not try to manage memory for shaders. Shaders get a block of memory for their use and input to the entry point function. Everything else is up to the shader. (i.e. if the shader wants to dma in or out memory, it's handled internally to the shader.)
4. Shaders should be called with as many instances of data as possible. (i.e. prefer shaders that iterate over arrays of data rather than a single instance or fragment.)

Example from RCF: igPhysics

[igPhysics was written by Eric Christensen]

The igPhysics system is the first system to make the transition to our next generation of SPU design. The result was physics that are both faster and simpler. What follows is an outline of the stages igPhysics had gone through as part of this upgrade:

STAGE 1: Dynamic Code Loading

The first thing that had to happen to make this system work was to be able to dynamically load in any fragment of SPU code completely on-demand.

SPURS, or any similar system, complicates what is in fact a very simple process. On the SPU code is data. There's nothing more to it than that. So while still using SPURS for igPhysics and other systems, we're not relying on any of the features (other than starting an initial job). At some point we will probably cull use of SPURS from the engine overall, but at the moment it's being used because it was already in-place.

- On SPU, Code is data
- Double buffer / stream same as data
- Very easy to do (No need for special libraries)
 - Compile the code
 - Dump the object as binary
 - Load binary as data
 - Jump to binary location (e.g. Normal function pointer)
- Pass everything as parameters, the ABI won't change.

For more specific details on how we, more or less, load the code see:

http://www.insomniacgames.com/tech/articles/0807/files/dynamic_spu_code.txt

For the igPhysics system, the shader interface is defined as:

```
typedef void (SPU_Printf) (const char *, ...);
typedef void (SPU_DmaPut) (void *ls, u32 ea, u32 size, u32 tag);
typedef void (SPU_DmaGet) (void *ls, u32 ea, u32 size, u32 tag);
```

```
// this is the common data that gets passed to the shader
```

```

struct Common
{
    SPU_Printf *m_printf;
    SPU_DmaPut *m_dma_put;
    SPU_DmaGet *m_dma_get;
    u8 *m_work_ls;

    u32 m_dma_tags[PHYSICS_SHADER_NUM_DMA_TAGS];

    u8 *m_input_data;
    u32 m_work_ls_size;
    u32 m_spu_id; // which spu is running this fragment
    u32 m_pad;
};

// Standard Shader Function
// parameters:
// Common (all the common functions)
// u32 (effective address of PPU data buffer being referenced by the shader)

typedef void (Shader) (Common *, u32);

```

STAGE 2: Dividing, and simplifying, the physics pipeline

Once dynamic code loading was working, igPhysics was reworked into a much simpler multistage pipeline where each stage loaded the next.

This is an example of how shaders are specifically NOT generic. Each pipeline stage is completely aware of the stage that follows (but not more) and loads explicitly with the minimum work necessary. But because the order is explicit and custom to this system, the fragment for each stage can keep or eject data from the SPU local store with knowledge of what the next will need.

While the overall stages are completely fixed, the individual code fragments within a stage aren't necessarily. Depending on the type of physics interaction being simulated, for example, a specific shader may need to be resident. The data is first sorted by similar type, then each type's code fragments are loaded on-demand. For example, the simulation stage has the following fragments which may be run:

```

pcf_build_jd_ball.cpp
pcf_build_jd_contact_point.cpp
pcf_build_jd_dual_hinge.cpp
pcf_build_jd_dual_hinge_with_limits.cpp
pcf_build_jd_hinge.cpp
pcf_build_jd_hinge_with_limits.cpp
pcf_build_jd_lock.cpp
pcf_build_jd_slider.cpp
pcf_build_jd_slider_with_limits.cpp
pcf_build_jd_spring.cpp
pcf_build_jd_wobble.cpp
pcf_collide_capsule_vs_capsule.cpp
pcf_collide_capsule_vs_obb.cpp
pcf_collide_capsule_vs_trimesh.cpp
pcf_collide_obb_vs_obb.cpp
pcf_collide_obb_vs_trimesh.cpp

```

```
pcf_collide_sphere_vs_capsule.cpp
pcf_collide_sphere_vs_obb.cpp
pcf_collide_sphere_vs_sphere.cpp
pcf_collide_sphere_vs_trimesh.cpp
```

STAGE 3: Added external system shaders

Once igPhysics was using shaders for all its internal functionality, it was time use the same idea to have it interact with the rest of the game. There are two main places where igPhysics needs to communicate with other systems:

1. Events need to be send to the Effects Conduit. Basically whenever anything interesting in physics happens, that event is sent to the Conduit which then forwards it on to any system that has a matching request.

In this case, the shader is basically the “glue” between the physics system and the conduit system. The physics system itself does not need to contain any conduit related information, nor does it need to have any code dependencies on the conduit. The dependencies are completely contained in the shader.

Also, where previously each event would have immediately made a call into the conduit (callback-style), now the events are cached and deferred – once all the physics objects are gone through, the list is iterated once and sent to the conduit. This has the benefit of both reducing synchronization and thrashing of the PPU instruction cache (the conduit is a PPU system).

2. Gameplay systems need to collect physics data and possibly modify it based on custom conditions.

The concept of these shaders is to allow Gameplay systems to affect physics data (change on-the-fly) while they are being evaluated based on specific Gameplay conditions. Once again, the physics system itself does not need to be aware of the changes or the dependencies on the Gameplay data and code. All of that is contained in the shader.

Benefits of SPU Shaders to igPhysics

- Pipeline well defined and completely SPU-driven
- SPU processing completely asynchronous
- Data well-organized and well-defined.
- No (or minimal) PPU intervention

EOF

