

SPU Physics

Eric Christensen
Principal Engine Programmer
Insomniac Games

Erwin Coumans
Simulation Support Lead
Sony Computer Entertainment America

Insomniac Physics System

Feel free to ask questions at any time.

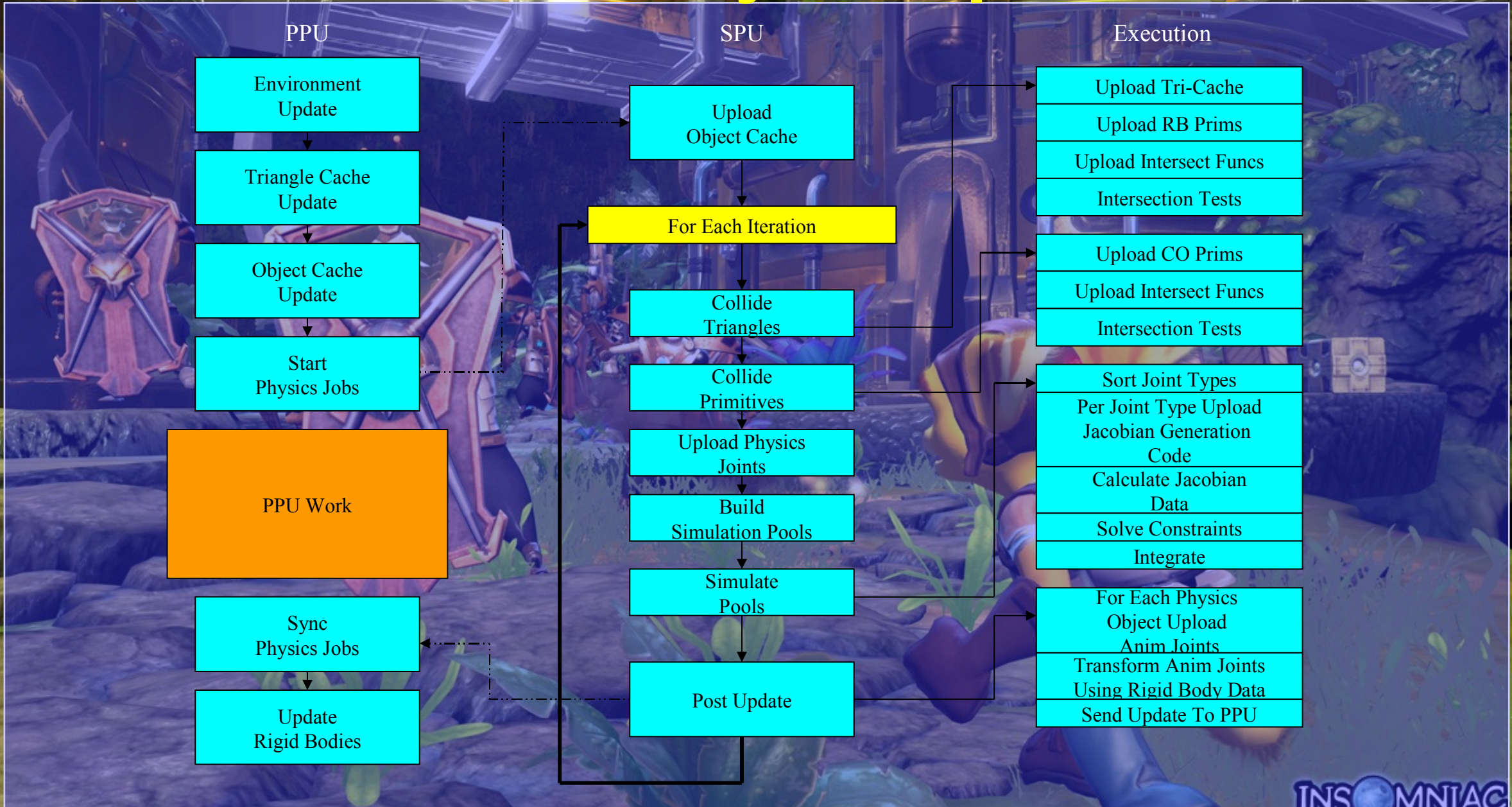
SPU Shaders

- Can be written for the physics system to do specific things without having to change the physics pipeline.
- Built individually, resulting code lives in a header file that contains a byte array.
- Debug/Read-only section resides at the bottom of the byte array.
- Shaders written by gameplay programmers are registered with the physics system through a simple API.
- The physics system uploads registered shaders for a particular context at a particular stage.
- 2k of local store is allocated for the shader code, 512 bytes is allocated for the shader work buffer.

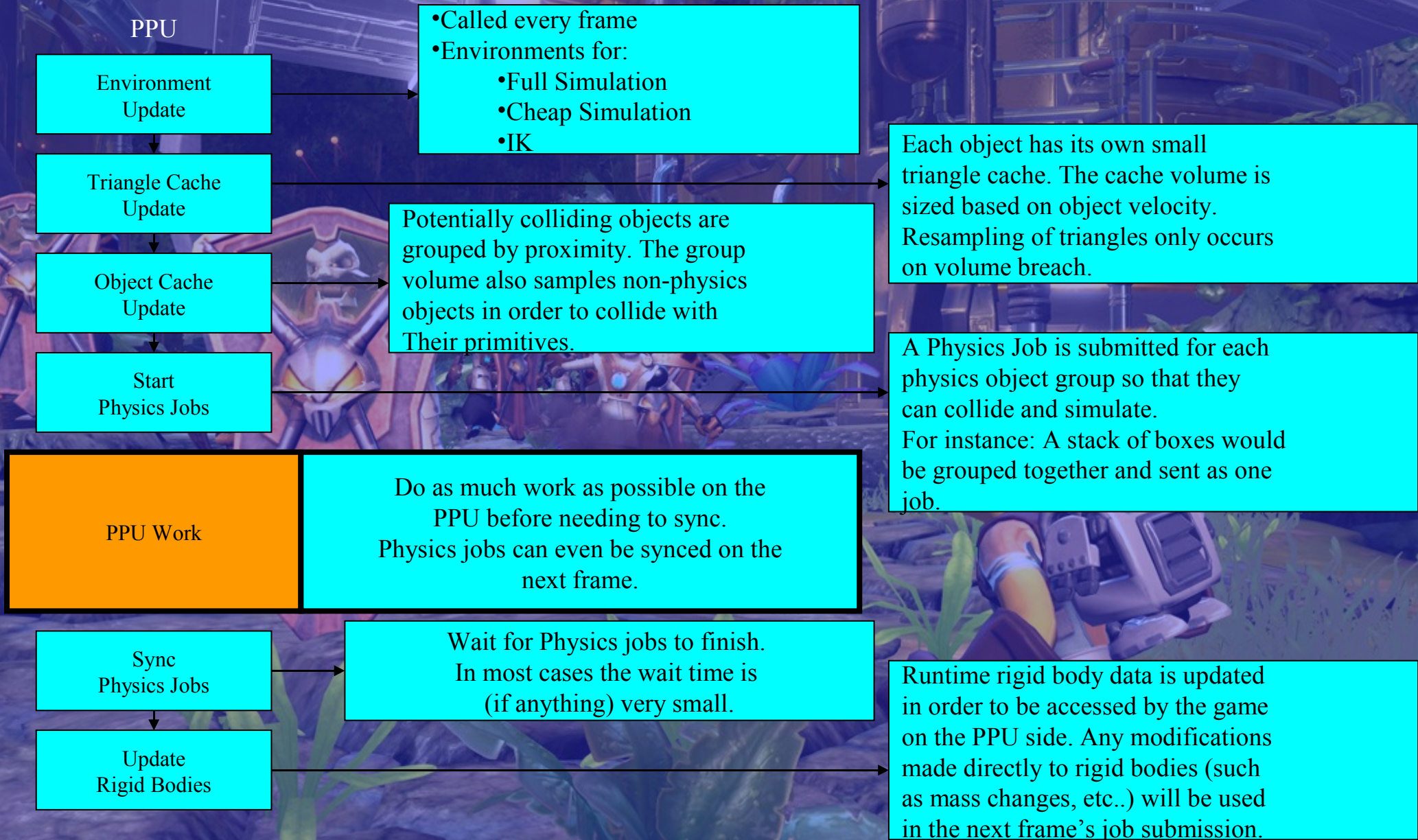
SPU Shaders

- Shaders access a common set of functions that are passed to them, as well as pre-allocated dma tags.
Some functions for example:
 - DMA get/put
 - Printf
 - Local store allocator
- Physics system has its own set of shaders that are native and expected in the pipeline, however, they are built the same way.
- Native physics system shaders are only slightly optimized currently and range from 2k to 12k in size.

Insomniac Physics Pipeline



Insomniac Physics Pipeline



Collide Triangles

Allocate local store for triangle cache.
Tri-cache size is fixed, so this only needs
to occur once.

For Each Physics Object (which can contain many rigid bodies i.e. Ragdolls)

Upload Tri-Cache

- Upload the corresponding triangle cache.
- It goes in the throw-away memory allocated for all tri-caches.

Upload Rigid Body Primitives

- For each physics object, upload the collision primitives for each rigid body contained in the object.
- Sort the primitives by type.

Upload Triangle Intersection Code

Perform Intersection Tests

- Local store is allocated as needed for intersection code.
- For each list of primitive types, upload the corresponding code.
i.e. Sphere, Capsule, OBB vs Trimesh
- Perform intersection and contact point generation.

Roll back local store to beginning of
Pre-allocated contact point buffer.

Collide Primitives

For Each Physics Object

Upload Rigid Body Primitives

- For each physics object, upload the collision primitives for each rigid body contained in the object.
- Sort the primitives by type.

Upload Triangle Intersection Code

Perform Intersection Tests

- Local store is allocated as needed for intersection code.
- For each list of primitive types, upload the corresponding code. i.e. Sphere vs. Sphere, Capsule vs. OBB, OBB vs OBB etc..
- Perform intersection and contact point generation.

Roll back local store to beginning of
Pre-allocated contact point buffer.

Upload Rigid Body Data & Joints

Allocate local store for all Rigid Bodies
and Joints

For Each Physics Object

Upload Rigid Bodies

Store Rigid Bodies data in its corresponding local store buffer.
Insert Rigid bodies into sort list nodes.

Upload Joints

Store Joint data in it's corresponding local store buffer.
Insert Joints into sort list nodes.

Build Simulation Pools

Associate Rigid Bodies

Each Joint (including contact points) has a relationship with one
or two Rigid Bodies.
Pools are built based on Rigid Body connectivity.
Each Joint is also inserted into it's respective simulation pool.

Simulate Pools

For each Simulation Pool “package”

Sort Joints by Type

- Joints are sorted based on constraint type.
i.e. Single Axis Hinge, Dual Axis Hinge, Spring, etc...
- The constraint type id references a table that describes the address and size of the corresponding Jacobian generation code.

Allocate local store for solver data.
Size is determined by number of Rigid Bodies in the pool
as well as the number of DOF removed from the system.

Generate Jacobian Data

- For each constraint type
 - Allocate local store for size of code fragment
 - Upload code fragment for constraint.
i.e. ContactPointBuildJD
- After generating Jacobian data, roll back local store to make room for next code fragment.

Solve Constraints

Iterative LCP solver to generate corrective forces.

Integrate

Update rigid body transform.

Post Update

For Each Physics Object

Upload Animation Joint data

- Allocate local store for animation data.
 - Matrix array
- Mapping of physics joints to animation joints

Transform Animation Joints

- Update animation joints that are mapped to rigid bodies using current Rigid Body transforms.
- This also includes updating “Game Actor” world-space matrix

DMA Results to PPU

- Send updated Rigid Body transforms.
- Send updated animation joints.
- Send updated “Game Actor” world-space matrix.

Rollback All Local Store to the end of the contact point buffer for the next iteration.

Summary Insomniac Physics

Limitations

- Limited Number of Collision Triangles per Physics Object
- Limited Number of Object Interactions per Job
- Limit on Number of Constraints per Job
- Limited Number of Contact Points per Job
- Not Possible to Run Collision and Simulation Separately

Benefits

- Discrete Pipeline Allows heavy Optimization
- No Intermediate Data Swapping Between PPU and SPU
- Physics Jobs Are Completely Independent
- Fixed PPU Memory Size Eliminates Dynamic Allocation from SPU
- Implemented and Optimized for a Specific Platform
- No Restrictions on Code Size
- Start and Sync Jobs Only Once Per Frame

Limitations

- **Limited Number of Collision Triangles per Physics Object**
Triangle Cache is a fixed size, therefore environment collision must be optimized to prevent exceeding the buffer size, which can result in thrown away geometry. This means that while the game is in development it's possible that objects might fall through the floor or pass through a wall if there are too many triangles to cache. However, we can make this type of assumption because some of our other systems rely on optimized environment collision as well
- **Limited Number of Object Interactions per Job**
Object Cache is a fixed size, meaning that there is a limitation on the amount of object to object interaction.
- **Limit on Total Number of Rigid Bodies per Physics Object**
The number of rigid bodies that a physics object can contain is limited by a fixed size. Cautious rigging of physics object is required as a result.

Limitations

- **Limited Number of Constraints per Job**

The maximum number of simultaneously solved constraints is limited to a fixed size. I.e. there is a certain limit on how many ragdolls (which on average are our most joint heavy objects) can be stacked by default. However there are tricks used to stack a virtually unlimited amount of objects, in which case some interactions can appear to be “non-physical”.

- **Limited Number of Contact Points per Job**

Number of contact points is fixed per job (contact points are generated every iteration). Because of our other limitations, this doesn't present a problem.

- **Not Possible to Run Collision and Simulation Separately**

Our Physics collision and simulation live together, so only using collision or simulation separately isn't possible.

Benefits

- **Discrete Pipeline Allows Heavy Optimization**
Because we know our limits, the data transformation functions don't have to be generalized, allowing for deep optimizations.
- **No Intermediate Data Swapping Between PPU and SPU**
Once a job is in flight, intermediate data doesn't have to be sent back to the PPU to make more room in local store because we've already pre-calculated our memory requirements. Additionally we take advantage of the fact that we can upload code and data whenever we want, freeing up valuable local store when we need it.
- **Physics Jobs Are Completely Independent**
Physics jobs aren't dependent on other physics jobs, therefore once a pool of objects has been created, we can send the job any time we want, and retrieve the results any time we want.
- **Fixed PPU Memory Size Eliminates Dynamic Allocation from SPU**
The system doesn't need to allocate more resources than it's prepared for, so the job never needs to request that a PPU thread allocate more memory in case something needs to be cached.

Benefits

- **Implemented and Optimized for a Specific Platform**

The system is designed around a specific platform (PS), and built around the limitations of the game engine. Any changes to the engine result in customized changes to the physics system, whether it be lifting restrictions, or adding more. Our pipeline is relatively simple so any system-wide changes are not that much of a hassle. (such as the addition of new joint types or collision functions). In the worst case scenario, we break up our functions even further to free up more local store for an increased data requirement.

- **No Restrictions on Code Size**

Since the majority of the code is “shader” based, we don’t have to take into consideration local store restrictions when adding more functions. I.e. adding 5 more collision or jacobian building functions that are 10k a piece on average makes no difference. Currently, all of the SPU code from the physics system exceeds 100k, but it doesn’t affect our local store limits.

- **Start and Sync Jobs Only Once Per Frame**

We only need to sync once since our physics collision and simulation live together (see limitations 😊), and we can sync anywhere we want.

Debugging Techniques

- Asserts for catchable cases, which print basic information about the crash. File, Line, Function, etc...
- Each function entry point is wrapped with a macro that inserts the Function name, File, Line into a buffer that is at a fixed address in local store.
- A buffer for each SPU exists on the PPU that is large enough to accommodate 4 levels of call-stack data.
- With TRAP_CALLSTACK enabled, at function entry, the UPDATE_CALLSTACK macro will call a stubbed function that uploads the current call-stack buffer from the PPU, modifies it, then sends it back.
- In the event of a crash, we time-out trying to sync on the PPU, which will result in dumping the call-stack data to the TTY.

Debugging Techniques

- In addition to keeping track of function calls, there is also an option to DMA function input data to the PPU in roughly the same manner as the call-stack.
- Since most of the functions are shaders, they can be built independently and run on a PS3 Linux box.
- In the event of a crash, function input data that has been sent to the PPU can now be used as input to the same function, which can now be debugged in isolation with the conditions that caused the crash.
- These techniques are still in their infancy but can be very useful in tracking down previously very painful bugs.

Bullet Physics System

Free open source SDK at <http://bulletphysics.com>
Contact Erwin Coumans for Playstation 3 optimized version.

Feel free to ask questions at any time.

Portable SPURS Tasks

- Porting the implementation for
 - Local Store memory
 - DMA transfers
 - Threading
 - Synchronization
- Supported threading environments:
 - PS3 Cell SPURS Task
 - PS3 Cell Libspe2 (IBM Cell Blade, PS3 Linux)
 - Windows/Xbox 360 Threads
 - etc

Bullet Physics Pipeline



Bullet Physics Pipeline

PPU

Apply Forces/
Predict Transform

- Predict transforms for swept / continuous collisions

Update
Overlapping Pairs

- Persistent overlapping Pair Cache
- Only track changes
- Perform Collision Filtering

Dispatch
Overlapping Pairs

- PPU allocates contact cache
- Each frame, points can be added, modified/removed

PPU Work

- PPU fallback for cases that are not implemented on SPU yet
- PPU collision callbacks

Sync
Collision Tasks

Build Islands &
Hash Cells

- Use spatial hash function to divide bodies & constraints into cells
- Build dependency table between cells, using constraint connectivity
- Use islands to determine (de)activation state

Solve
Constraints

PPU Work

Sync
Physics Tasks

Integrate

- Option to use time of impact to avoid missing collisions (for fast / small moving objects)

Update
Rigid Bodies

Collide Triangles

- Interleave search with triangle-intersection test
- All code fits in SPU memory

For each Collision Pair involving Triangle Meshes

Upload Tree Parts

- Fast acceleration structure to find bounding volume overlap
- Only DMA Tree Part if necessary

Traverse AABB Tree

- Perform stackless tree traversal

Upload Triangle

- DMA the 3 triangle indices and vertices
- Use software cache re-use shared vertices

Perform Intersection Test

- Perform a triangle versus convex primitive intersection test
- Generic GJK Intersection test reduces code size
- Throw away triangle after intersection test

DMA contact point caches from/to main memory

Collide Primitives

For Each Overlapping Pair of Convex Shapes

Upload Collision Pair

Upload Transforms

Upload Collision Shape

Upload Contact Cache

Generic GJK Collision

Send Contact Cache

- Most primitive collision shapes are fixed size
- Convex meshes need additional temporary storage

- Each combination of convex shape uses generic algorithm
- Sphere, OBB, Capsule, Cylinder, Cone, Convex Hull etc..
- For deep penetrations use simplified algorithm that reduces code size and memory usage
- Perform contact point cache reduction on SPU

DMA contact point caches from/to main memory

Pack Rigid Body Data & Setup Constraints

Stream Rigid Bodies and Constraints

For Each Simulation Cell

Upload Rigid Body

Setup Solver Body

Send Bodies

For each Rigid Body in the Cell, create a packed Solver Body
Keep a mapping between them, by index

Upload Contact Caches

Upload Joints

Prepare Constraints

Send Packed Constraints

- Pack each joint in the Cell, referencing the packed solver body
- Build Jacobian data for each constraint

DMA Packed Bodies and Constraints to main memory

Simulate Cells

For Each Solver Iteration

Synchronize SPUs

- Make sure each SPU is processing the same solver iteration
- Use spinlock to wait for other SPUs

Find Next Cell

- Find the next available Cell in main memory
- Use Dependency Table to avoid conflicts with other SPUs
- Update Cell status

Upload Packed Data

Solve Constraints

Send Packed Data

- Upload Packed bodies and constraints for this Cell
- Perform one solver iteration on all constraints
- Send packed bodies and constraints to main memory

Send Updated Body Velocities to main memory

Summary Bullet Physics

Limitations

- Code has to fit entirely in SPU memory
- Generic convex collision algorithm performs a bit less than specific
- Requires Intermediate Data Swapping Between PPU and SPU
- Generic approach requires synchronization between SPUs
- Contact points need to be stored in main memory

Benefits

- Multi-platform makes development and debugging easy
- SPURS Task ports well to Win32/Xbox 360 threads (fits L2 cache)
- Easy to extend with new custom collision algorithms
- No Restrictions on Data Sizes
- Automatic broadphase allows arbitrary environments
- Fixed PPU Memory Size Eliminates Dynamic Allocation from SPU
- Flexible use of Collision and Simulation Separately

Limitations

- **Code has to fit entirely in SPU memory**

Although we carefully chose compact algorithms, the 256kb size restriction can hinder development. Debugging the SPU can be a challenge (only add debug information for some files). SPU Shaders or overlays would help.

- **Generic convex collision algorithm performs a bit less then specific**

Especially for very simple cases like sphere versus sphere, the computational cost is higher. However, for more complex cases, the cost is closer to an optimized special case.

Limitations

- **Generic approach requires Intermediate Data Swapping Between PPU and SPU and additional synchronization**

Dealing with unrestricted number of collision shapes, rigid bodies and constraints cost performance. However, hiding DMA latency and parallel processing on SPUs still gives good performance.

- **Contact points need to be stored in main memory**

The generic convex collision detection only computes one contact point at a time. Stable stacking in rigid body dynamics requires multiple contact points. Our current solution is to maintain a persistent contact cache and add, refresh, remove contact points. Other solutions exist, that compute all contact points at a time. On the flipside, keeping contact points allows to store additional information that can improve constraint solver quality (warm starting), and easier access to the forces/impulses applied by the solver.

Benefits

- **Multi-platform makes development and debugging easy**

At any stage of development it is possible to debug or prototype improvements on other platforms. Sharing a common code-base reduces maintenance.

- **SPURS Task ports well to Win32/Xbox 360 threads**

Porting a SPURS Task to Win32/XBox360 can be very smooth. The data access is cache friendly. Each thread creates its own local copy of the Local Store memory. To avoid a memcpy, a read-only DMA returns the original address.

- **Easy to extend with new custom collision algorithms**

Instead of creating N new collision algorithms for each collision type, we only need to implement a single support function. We can also add or improve additional collision shapes using combinations of compounds, scaling, translation and rotation etc..

Benefits

- **No Restrictions on Data Sizes**

Cell SPUs have enough power to deal with many interacting rigid bodies. Our dynamic splitting method using dependencies simulates extremely large simulation islands correctly. This means the SPUs can handle huge concave triangle meshes with local deformation.

- **Automatic broadphase allows arbitrary environments**

Large outdoor environment or dynamic changing environments can be simulated efficiently using the incremental broadphase. This approach doesn't require any manual partitioning.

- **Fixed PPU Memory Size Eliminates Dynamic Allocation from SPU**

We allocate all memory on the PPU and avoid dynamic allocations using memory pools and stack allocators.

- **Flexible use of Collision and Simulation Separately**

The SPU collision detection system can be re-used for synchronous collision queries by the AI system or game logic. For example batches of ray tests for line-of-sight, swept collision checks for player movement or camera positioning etc.

Debugging Techniques

- Since the SPURS Task also runs multiplatform, we can debug under Windows
 - Easier to detect Data and DMA alignment issues
 - Avoid forking implementation
- Since the SPURS Task also runs under Libspe2, we can prototype in a very similar environment on a PS3 Linux box
- Asserts for catchable cases, which print basic information about the crash. File, Line, Function, etc...

ec@insomniacgames.com
Insomniac Games

erwin_coumans@playstation.sony.com
Sony Computer Entertainment America