GDC 13

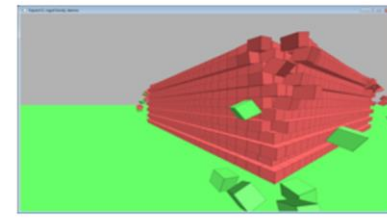
GPU Rigid Body Simulation

Erwin Coumans
Principal Engineer @ http://bulletphysics.org

GAME DEVELOPERS CONFERENCE
SAN FRANCISCO, CA
MARCH 25-29, 2013
EXPO DATES: MARCH 27-29
2013

Erwin Coumans

- Leading the Bullet Physics SDK project http://bulletphysics.org
- Doing GPGPU physics R&D at AMD, open source at http://github.com/erwincoumans/experiments
- Previously at Sony SCEA US R&D and Havok



Welcome to this talk about GPU Rigid Body Dynamics. My name is Erwin Coumans and I am leading the Bullet Physics SDK project. I have been working on Bullet 100% of my time for about 10 years now. First as a Sony Computer Entertainment US R&D employee and currently as AMD employee.

GPU Cloth (2009)

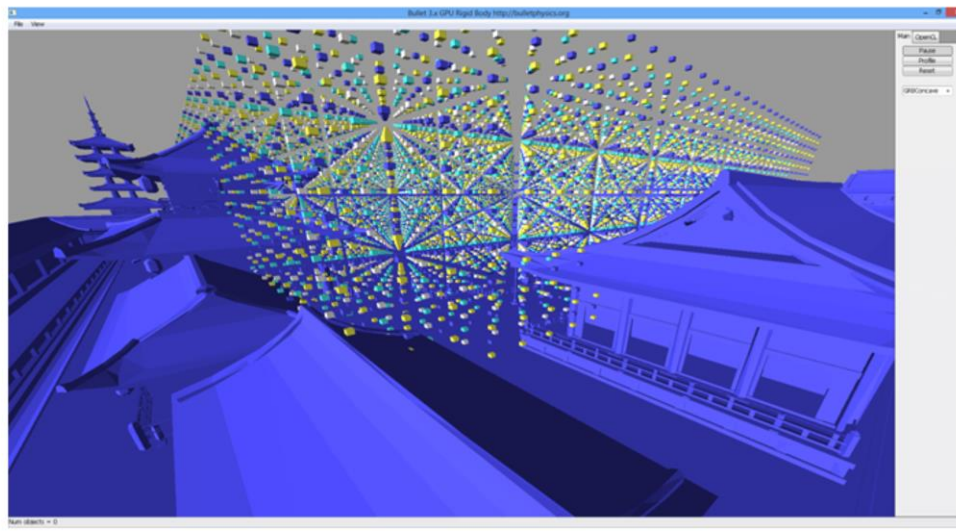
Our initial work on GPGPU acceleration in Bullet was cloth simulation. AMD contributed OpenCL and DirectX 11 DirectCompute accelerated cloth simulation. This is available in the Bullet 2.x SDK at http://bullet.googlecode.com

Here is a screenshot of the Samurai Demo with Bullet GPU cloth integrated.

GPU Hair (2012/2013)

More recently one of the engineers in my team worked on GPU accelerated hair simulation. This work was presented at the Vriphys 2012 conference by Dongsoo Han and integrated in the latest Tombraider 2013 game under the name TressFX.

GPU Rigid Body (2008-2013)



We started exploring GPU rigid body simulation at Sony Computer Entertainment US R&D. My SCEA colleague Roman Ponomarev started the implementation in CUDA and switched over to OpenCL once the SDKs and drivers became available. I'll briefly discuss the first generation GPU rigid body pipeline.

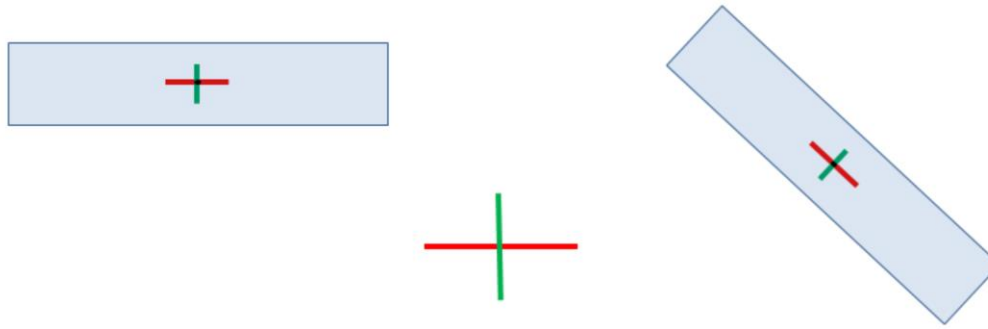
When I joined AMD, my colleague Takahiro Harada implemented most of the second generation GPU rigid body pipeline, around 2010.

From 2011 onwards I'm working on the third generation GPU rigid body pipeline, with pretty much the same quality as the Bullet 2.x CPU rigid body simulation. This is work in progress, but the results are already useful.

One of the GPU rigid body demos uses a simplified triangle mesh of the Samurai Cloth demo, as you can see in this screenshot. The triangle mesh is reduced to around 100.000 triangles (there is no instancing used).

Rigid Bodies

- Position (Center of mass, float3)
- Orientation (Inertia basis frame, float4)

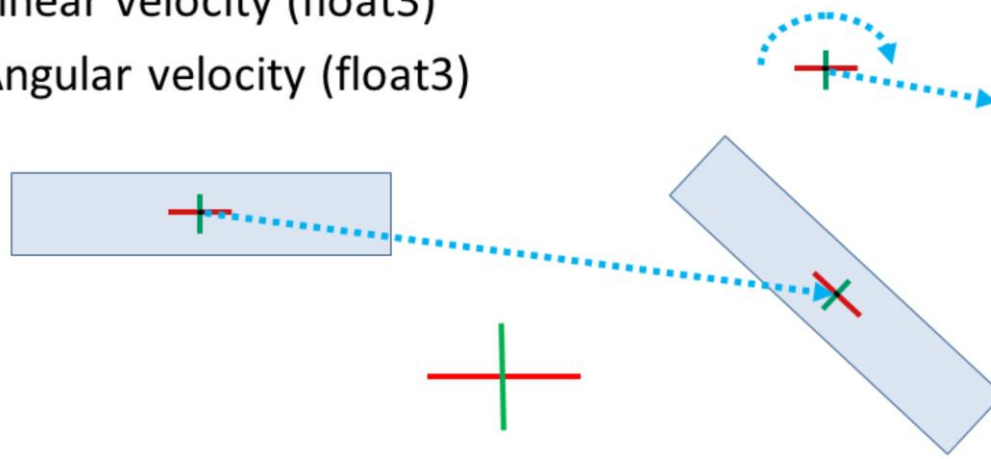


Before going into details, the most essential parts of a rigid body are the world space position and orientation of rigid bodies. In our case the position defines the center of mass, using a float3. A 4rd unused w-component is often added to make this structure 16 byte aligned. Such alignment can help SIMD operations.

The orientation defines the inertial basis frame of a rigid body and it can be stored as a 3x3 matrix or as a quaternion. For Bullet 2.x we used a 3x3 matrix, but in Bullet 3.x we switched over to quaternions because it is more memory efficient.

Updating the transform

- Linear velocity (float3)
- Angular velocity (float3)



The rigid body bodies are simulated forward in time using the linear and angular velocity, using an integrator such as Euler integration. Both linear and angular velocity is stored as a 3-component float.

Update Position in C/C++

```
void  integrateTransformsKernel(Body* bodies, int nodeID, float timeStep)
{
                if( bodies[nodeID].m_invMass != 0.f)
                {
                                bodies[nodeID].m_pos += bodies[nodeID].m_linVel * timeStep;                    //linear velocity
                }
}
```

If we only look at the position and linear velocity, the C/C++ version is just a one liner. This would be sufficient for most particle simulations.

Update Position in OpenCL™

```
__kernel void  integrateTransformsKernel( __global Body* bodies,const int numNodes, float timeStep)
{
                int nodeID=get_global_id(0);
                if( nodeID<numNodes && (bodies[nodeID].m_invMass!=0.f))
                {
                                bodies[nodeID].m_pos += bodies[nodeID].m_linVel * timeStep;                    //linear velocity
                }
}
```

See opencl/gpu_rigidbody/kernels/integrateKernel.cl

As you can see the OpenCL version of this kernel is almost identical to the C/C++ version in previous slide. Instead of passing the body index through an argument, the OpenCL work item can get its index using the get_global_id() API.

Apply Gravity

```
__kernel void  integrateTransformsKernel( __global Body* bodies,const int numNodes, float timeStep, float angularDamping, float4 gravityAcceleration)
{
                int nodeID=get_global_id(0);
                if( nodeID<numNodes && (bodies[nodeID].m_invMass!=0.f))
                {
                                bodies[nodeID].m_pos += bodies[nodeID].m_linVel * timeStep;                  //linear velocity
                                bodies[nodeID].m_linVel += gravityAcceleration * timeStep;                  //apply gravity
                }
}
```

See opencl/gpu_rigidbody/kernels/integrateKernel.cl

We can apply the gravity in this kernel as well. Alternatively the gravity can be applied inside the constraint solver, for better quality.

Update Orientation

```
__kernel void  integrateTransformsKernel( __global Body* bodies,const int numNodes, float timeStep, float angularDamping, float4 gravityAcceleration)
{
	int nodeID=get_global_id(0);
	if( nodeID<numNodes && (bodies[nodeID].m_invMass!=0.f))
	{
		bodies[nodeID].m_pos+= bodies[nodeID].m_linVel*timeStep;		//linear velocity
		bodies[nodeID].m_linVel+=gravityAcceleration*timeStep;		//apply gravity
		float4 angvel=bodies[nodeID].m_angVel;		//angular velocity
		bodies[nodeID].m_angVel*=angularDamping;		//add some angular damping
		float4 axis;
		float fAngle=native_sqrt(dot(angvel, angvel));
		if(fAngle*timeStep>BT_GPU_ANGULAR_MOTION_THRESHOLD)		//limit the angular motion
			fAngle=BT_GPU_ANGULAR_MOTION_THRESHOLD/timeStep;
		if(fAngle<0.001f)
			axis=angvel*(0.5f*timeStep-(timeStep*timeStep*timeStep)*0.020833333333f*fAngle*fAngle);
		else
			axis=angvel*(native_sin(0.5f*fAngle*timeStep)/fAngle);
		float4 dorn=axis;
		dorn.w=native_cos(fAngle*timeStep*0.5f);
		float4 orn0=bodies[nodeID].m_quat;
		float4 predictedOm=quatMult(dorn, orn0);
		predictedOm=quatNorm(predictedOm);
		bodies[nodeID].m_quat=predictedOm;		//update the orientation
	}
}
```

See opencl/gpu_rigidbody/kernels/integrateKernel.cl

When we introduce angular velocity and the update of the orientation, the code becomes a bit more complicated. Here is the OpenCL kernel implementation, with identical functionality to the Bullet 2.x position and orientation update.

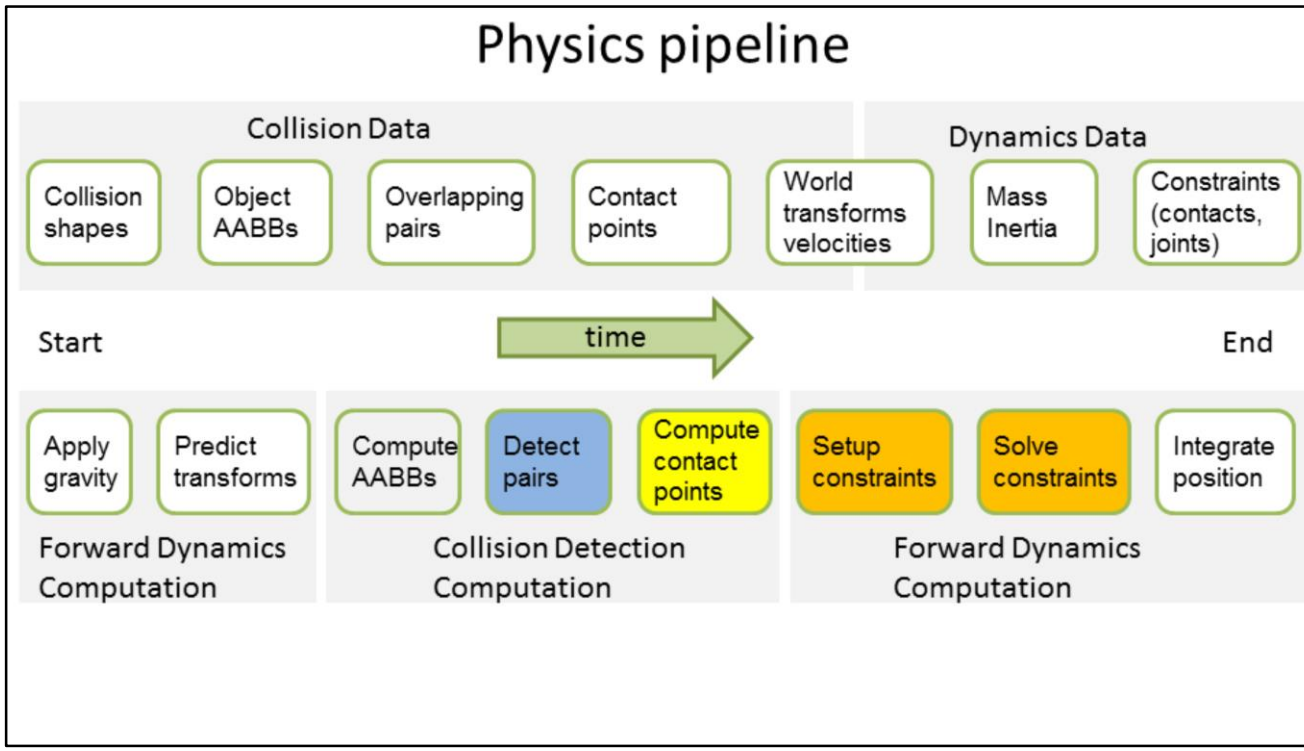
We have the option to apply some damping and to clamp the maximum angular velocity. For improved accuracy in case of small angular velocities, we use a Taylor expansion to approximate the sinus function.

Update Transforms, Host Setup

```
ciErrNum = clSetKernelArg(g_integrateTransformsKernel, 0, sizeof(cl_mem), &bodies);
ciErrNum = clSetKernelArg(g_integrateTransformsKernel, 1, sizeof(int), &numBodies);
ciErrNum = clSetKernelArg(g_integrateTransformsKernel, 1, sizeof(float), &deltaTime);
ciErrNum = clSetKernelArg(g_integrateTransformsKernel, 1, sizeof(float), &angularDamping);
ciErrNum = clSetKernelArg(g_integrateTransformsKernel, 1, sizeof(float4), &gravityAcceleration);

size_t workGroupSize = 64;
size_t   numWorkItems = workGroupSize*((m_numPhysicsInstances + (workGroupSize)) / workGroupSize);
if (workGroupSize>numWorkItems)
        workGroupSize=numWorkItems;
ciErrNum = clEnqueueNDRangeKernel(g_cqCommandQue, g_integrateTransformsKernel, 1, NULL, &numWorkItems, &workGroupSize,0,0,0);
```

Here is the host code to setup the arguments and the call to execute the kernel on the OpenCL device.

If we would have no constraints or collisions, we would be finished with our rigid body simulation,



Once we add collision detection and constraint solving a typical discrete rigid body simulation pipeline looks as in this slide. The white stages below are trivial to parallelize, or embarrassingly parallel, so we don't spend further time on it in this presentation.

The detection of potential overlapping pairs in blue, the contact point generation and reduction in yellow and constraint solving stages in orange are the main stages that we need to deal with.

All 50 OpenCL™ kernels

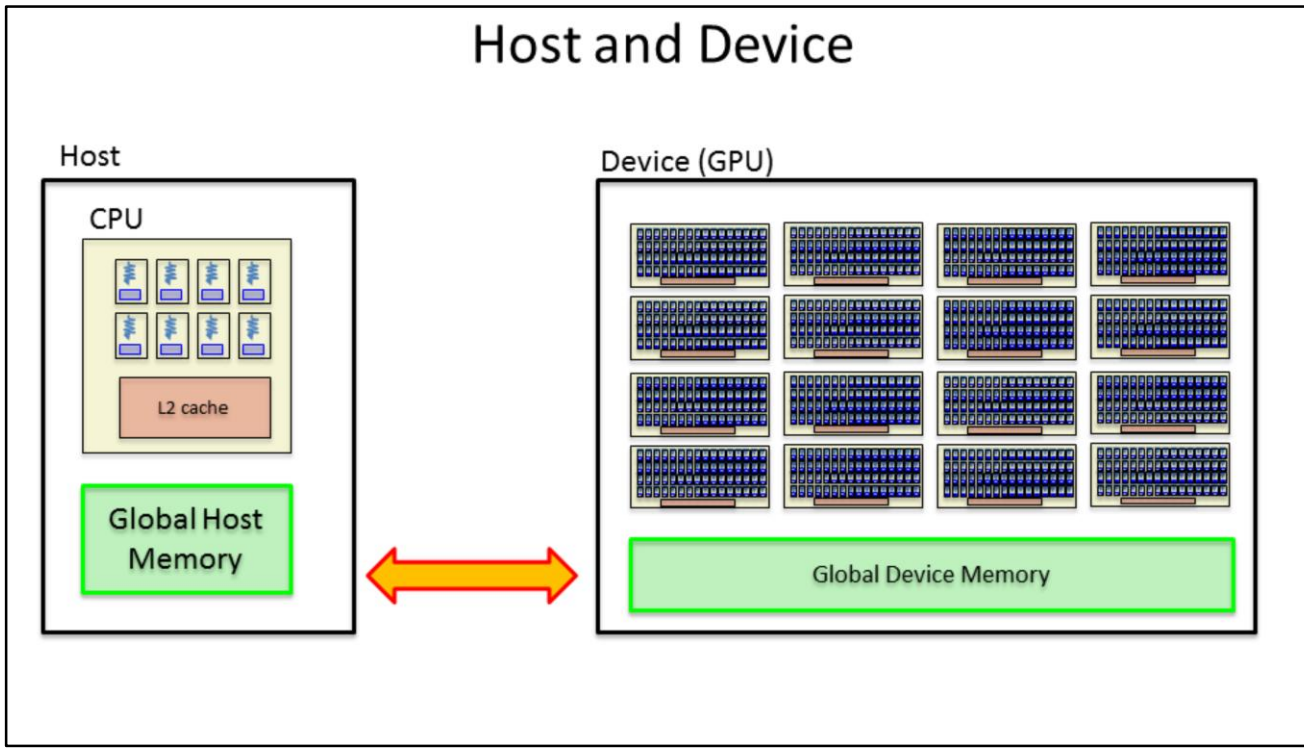
AddOffsetKernel	AverageVelocitiesKernel	BatchSolveKernelContact	BatchSolveKernelFriction	ClearVelocitiesKernel	ContactToConstraintKernel
ContactToConstraintSplitKernel	CopyConstraintKernel	CountBodiesKernel	CreateBatches	CreateBatchesNew	FillFloatKernel
FillInt2Kernel	FillIntKernel	FillUnsignedIntKernel	LocalScanKernel	PrefixScanKernel	ReorderContactKernel
SearchSortDataLowerKernel	SearchSortDataUpperKernel	SetSortDataKernel	SolveContactJacobiKernel	SolveFrictionJacobiKernel	SortAndScatterKernel
SortAndScatterSortDataKernel	StreamCountKernel	StreamCountSortDataKernel	SubtractKernel	TopLevelScanKernel	UpdateBodyVelocitiesKernel
bvhTraversalKernel	clipCompoundsHullHullKernel	clipFacesAndContactReductionKernel	clipHullHullConcaveConvexKernel	clipHullHullKernel	computePairsKernel
computePairsKernelTwoArrays	copyAabbsKernel	copyTransformsToVBOKernel	extractManifoldAndAddContactKernel	findClippingFacesKernel	findCompoundPairsKernel
findConcaveSeparatingAxisKernel	findSeparatingAxisKernel	flipFloatKernel	initializeGpuAabbsFull	integrateTransformsKernel	newContactReductionKernel
processCompoundPairsKernel	scatterKernel				

Here is an overview of the 50 kernels of our current third generation GPU rigid body pipeline. Note that 100% of the rigid body simulation is executed on the OpenCL GPU/device.

The green kernels are general useful parallel primitives, such as a parallel radix sort, a prefix scan and such. Often they are provided by a hardware vendor, such as AMD, Intel or NVIDIA. Bullet includes its own implementation for parallel primitives, that are reasonably well optimized to work well on most OpenCL GPU devices.

The kernels in blue are related to finding the potential overlapping pairs, also known as broadphase collision detection. The yellow kernels are related to contact computation and reduction, also known as narrowphase collision detection, as well as bounding volume hieraries to deal with triangle meshes, often called midphase. The constraint solver kernels are colored orange.

There is still more work to be done in various areas, such as supporting more collision shape types and constraint types aside from contact constraints.

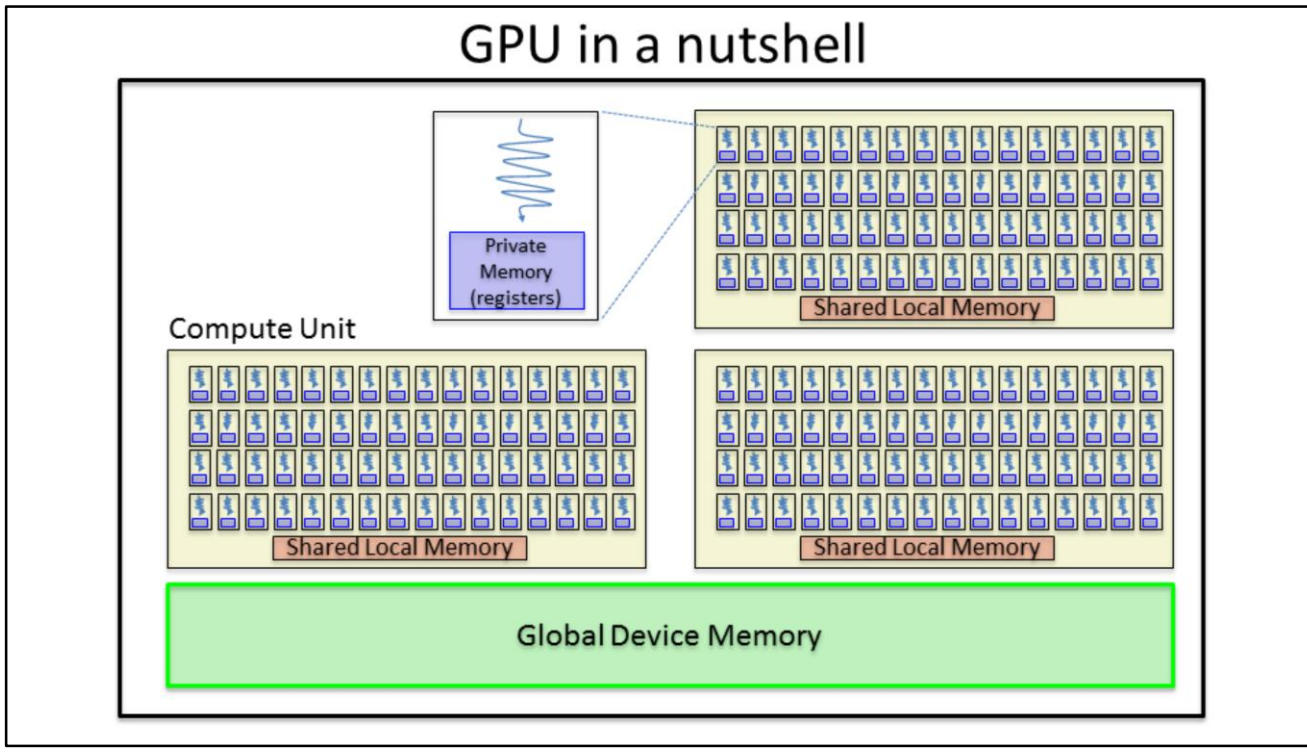


Before going into details how we optimized the various rigid body pipeline stages, here is a quick overview of a system with Host and Device.

Generally the host will upload the data from Global Host Memory to Global Device memory, upload the compiled kernel code to the device and enqueue the work items/groups so that the GPU executes them in the right order.

The connection between host and device, here the orange arrow, is a PCI express bus in current PCs. This connection will improve a lot in upcoming devices.

When using an integrated GPU we the global memory might be shared and unified between host and device.



A typical GPU device contains multiple Compute Units, where each Compute Unity can execute many threads, or work items, in parallel. For an AMD GPU there are usually 64 threads executed in parallel for each Compute Unit, we call this a wavefront. On NVIDIA GPUs there are usually 32 threads or work items executed in parallel for each Compute Unity, they call this parallel group of 32 threads a warp.

Each thread, or work item, has private memory. Private Memory on a 7970 is a 64kb register file pool, each work item gets some registers allocated.
Each Compute Unit has shared local memory, that can accessed by all threads in the Compute Unit. Shared Local Memory or LDS in AMD terminology, is 64kb on a 7970. Local atomics can synchronize operations between threads in a Compute Unit on shared local memory

The Global Device Memory can be access by all Compute Units. You can use global atomic operations on Global Device Memory.

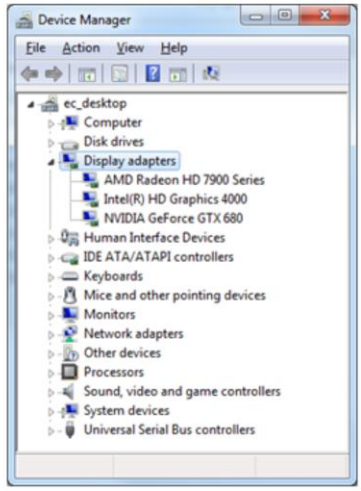
Shared Local Memory is usually an order of magnitude faster than global device memory, so it is important to make use of this for an efficient GPU implementation.

The programmer will distribute work into work groups, and the GPU has the responsibility to map the Work Groups to Compute Units. This is useful to make the solution scalable: we can process our work groups on small devices such as cell phone

with just a single or few Compute Units, or we can execute the same program on a high-end discrete GPU with many Compute Units.

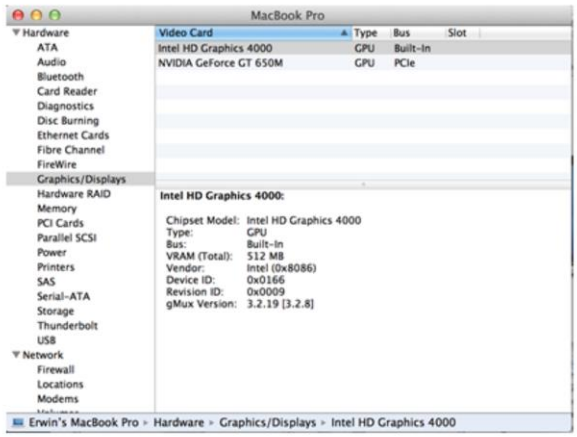
Windows GPU and CPU OpenCL Devices

- Support for AMD Radeon, NVIDIA and Intel HD4000



Here is a quick view of the GPU hardware as presented by the operating system. On the right we see a more detailed view from the GPU hardware through the OpenCL API. We can make use of the low level information to optimize our program.

Apple Mac OSX OpenCL Devices



The same information is available on Mac OSX or Linux or other OpenCL devices.

Other GPGPU Devices

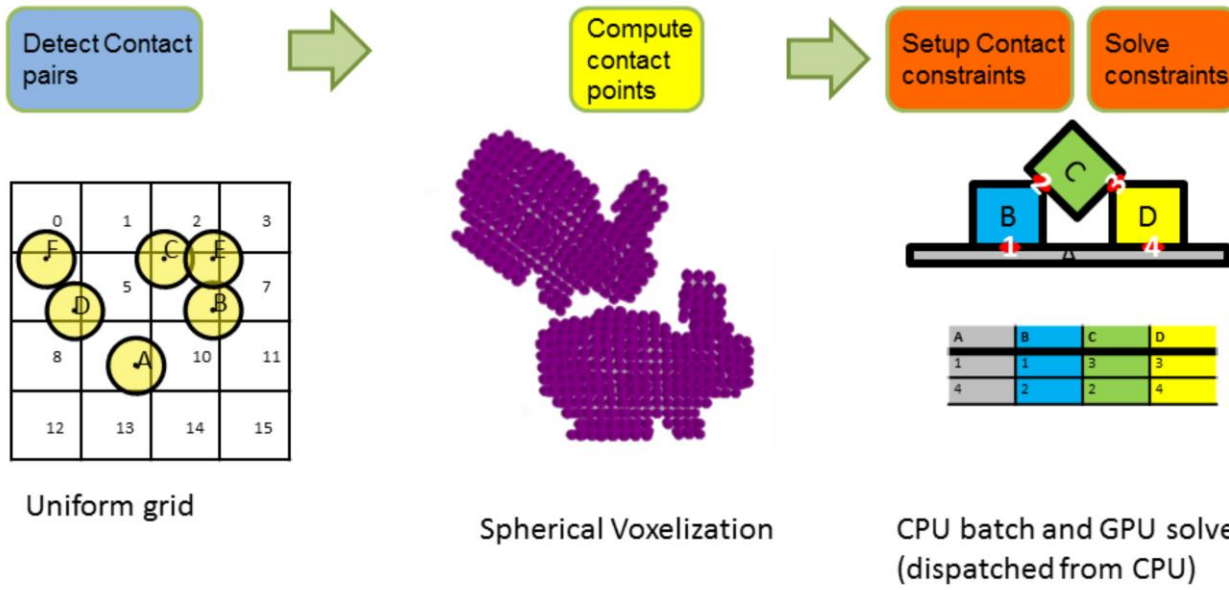
- Nexus 4 and 10 with ARM OpenCL SDK
- Apple iPad has a private OpenCL framework
- Sony Playstation 4 and other future game consoles

It appears that some OpenCL implementation is available as “easter egg” as parts of some OpenGL ES drivers in Android mobile devices such as the Nexus 4 or Nexus 10.

The Playstation 4 seems also suitable for GPGPU development: The system is also set up to run graphics and GPGPU computational code in parallel, without suspending one to run the other.

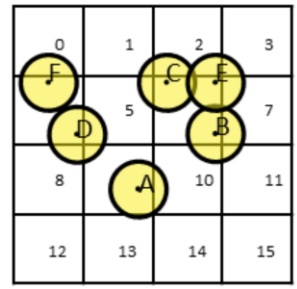
Chris Norden says that Sony has worked to carefully balance the two processors to provide maximum graphics power of 1.843 teraFLOPS at an 800Mhz clock speed while still leaving enough room for computational tasks. The GPU will also be able to run arbitrary code, allowing developers to run hundreds or thousands of parallelized tasks with full access to the system's 8GB of unified memory.

1st GPU rigid body pipeline (~2008-2010)



Here is the first generation of our GPU rigid body pipeline. A lot of simplification is used, so we could re-use a lot of work that was already available in GPU particle simulation.

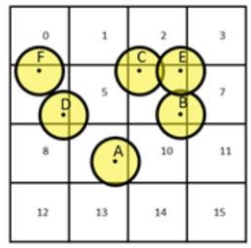
Uniform Grid



- Particle is also its own bounding volume (sphere)
- Each particle computes its cell index (hash)
- Each particle iterates over its own cell and neighborns

The uniform grid as already available in some OpenCL sdks, so we used this as a starting point for the GPU rigid body pipeline.

Uniform Grid and Parallel Primitives



Cell Index	Cell Start
0	
1	
2	
3	
4	0
5	
6	2
7	
8	
9	5
10	
11	
12	
13	
14	
15	

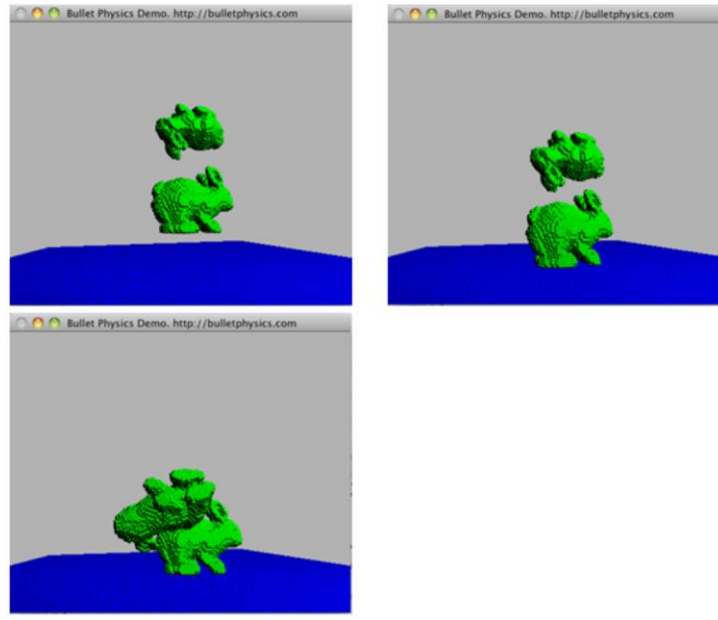
Array Index	Unsorted Cell ID, Particle ID	Sorted Cell ID Particle ID
0	9, A	4,D
1	6,B	4,F
2	6,C	6,B
3	4,D	6,C
4	6,E	6,E
5	4,F	9,A

- Radix Sort the particles based on their cell index
- Use a prefix scan to compute the cell size and offset
- Fast OpenCL and DirectX11 Direct Compute implementation

We use parallel primitives a lot when implementing GPU friendly algorithms, for example a parallel radix sort and prefix scan is used in a parallel uniform grid solution.

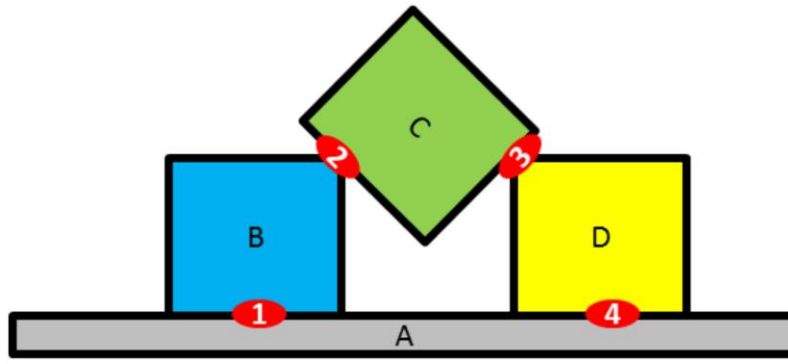
Contact Generation

The contact point generation was also simplified a lot, by approximating collision shapes using a collection of spheres. The process of creating a sphere collection, or voxelization, could be performed on CPU or GPU.



Here are some screenshots of a simulation of some Stanford bunnies dropping on a plane.

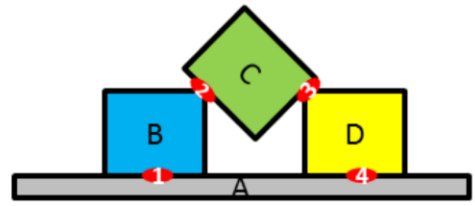
Constraint Generation



We use an iterative constraint solver based on Projected Gauss Seidel. In a nutshell, the velocity of bodies needs to be adjusted to avoid violations in position (penetration) and velocity. This is done by sequential application of impulses.

Pairwise constraints, such as contact constraints between two bodies need to be solved sequentially in a Gauss Seidel style algorithm, so that the most up-to-date velocity is available for each constraint.
An alternative would be to use Jacobi style constraint solving, where all constraints use “old” velocities, so we can trivially parallelize it. The drawback is that Jacobi doesn’t converge as well as Gauss Seidel.

Reordering Constraints



A	B	C	D
1	1		
	2	2	
		3	3
4			4

	A	B	C	D
Batch 0	1	1	3	3
Batch 1	4	2	2	4

- Also known as Graph Coloring or Batching

We cannot trivially update the velocities for bodies in each constraint in parallel, because we would have write conflicts. For example the contact constraint 1 in the picture tries to write velocity of object B (and A), while contact constraint 2 tries to update the same body B (and C). To avoid such write conflicts, we can sort the constraints in batches, also known as graph coloring. All the constraints in each batch are guaranteed not to access the same dynamic rigid body.

Note that non-movable or static rigid bodies, with zero mass, can be ignored: their velocity is not updated.

CPU sequential batch creation

```
while( nIdxSrc ) {
    nIdxDst = 0;    int nCurrentBatch = 0;
    for(int i=0; i<N_FLG/32; i++) flg[i] = 0; //clear flag
    for(int i=0; i<nIdxSrc; i++)    {
        int idx = idxSrc[i];  btAssert( idx < n );
        //check if it can go
        int aIdx = cs[idx].m_bodyAPtr & FLG_MASK;   int bIdx = cs[idx].m_bodyBPtr & FLG_MASK;
        u32 aUnavailable = flg[ aIdx/32 ] & (1<<(aIdx&31));u32 bUnavailable = flg[ bIdx/32 ] & (1<<(bIdx&31));
        if( aUnavailable==0 && bUnavailable==0 )   {
            flg[ aIdx/32 ] |= (1<<(aIdx&31));   flg[ bIdx/32 ] |= (1<<(bIdx&31));
            cs[idx].getBatchIdx() = batchIdx;
            sortData[idx].m_key = batchIdx; sortData[idx].m_value = idx;
            nCurrentBatch++;
            if( nCurrentBatch == simdWidth ) {
               nCurrentBatch = 0;
               for(int i=0; i<N_FLG/32; i++) flg[i] = 0;
            }
        }
        else  {
            idxDst[nIdxDst++] = idx;
        }
    }
    swap2( idxSrc, idxDst ); swap2( nIdxSrc, nIdxDst );
    batchIdx ++;
}
```

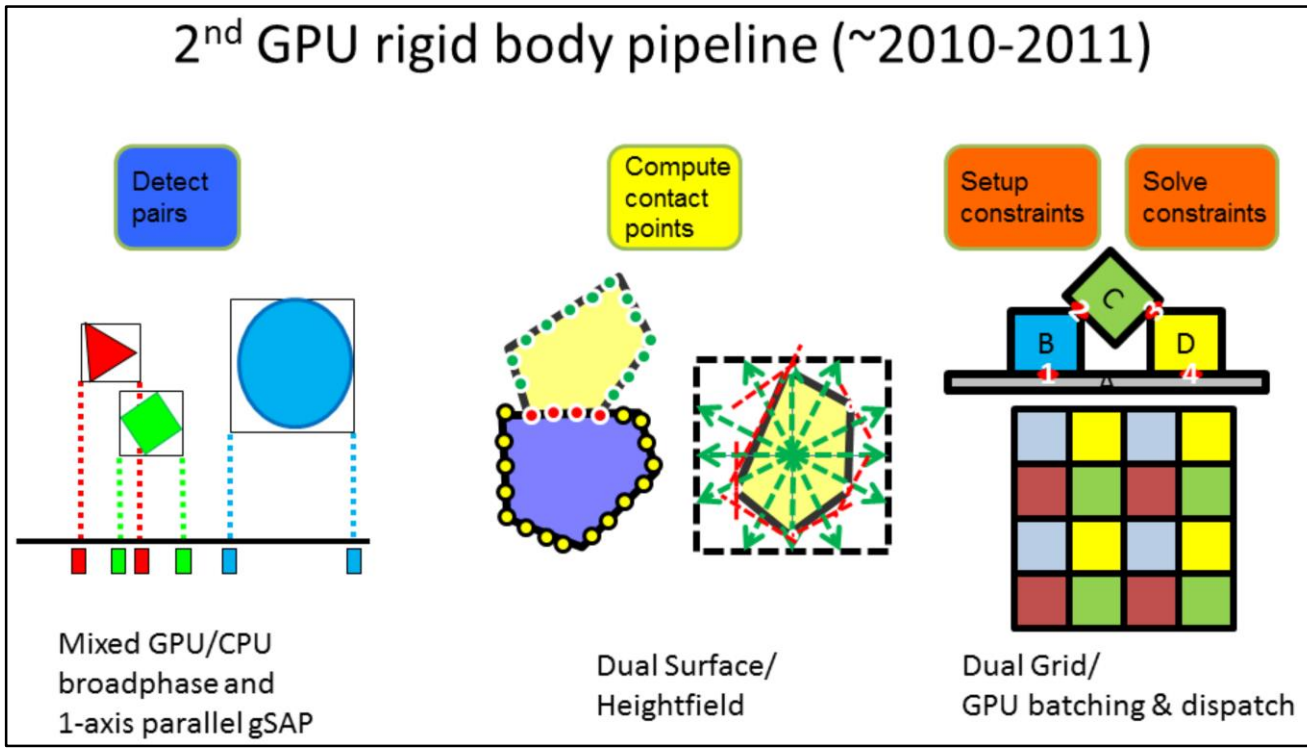
We first implemented the graph coloring or batch creation on the CPU host side. The code is very simple, as you can see. We iterate over all unprocessed constraints, and keep track of the used bodies for this batch. If the two bodies in a constraint are still available for this batch, we assign the batch index to the contraints. Otherwise the constraint will be processed in another batch.

There is a small optimization that we can clear the used bodies when we reach the device SIMD width, because only 32 or 64 threads are processed in parallel on current GPU OpenCL devices.

Naïve GPU batch creation

- Use a single Compute Unit
- All threads in the Compute Unit synchronize the locking of bodies using atomics and barriers
- Didn't scale well for larger scale simulations (>~30k)

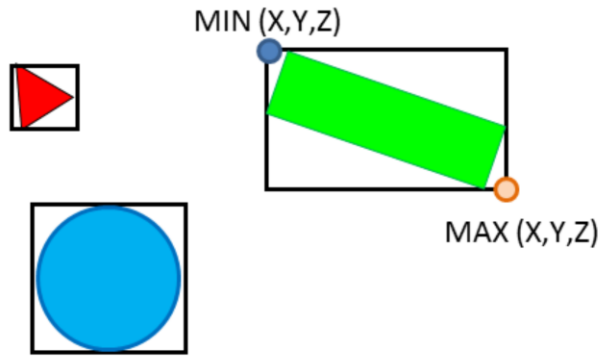
We can execute the same code on the GPU in a single Compute Unit, and use local atomics to synchronize between the threads. It is not optimal, but a good starting point. In a later pipeline we have improved the batch creation.



The uniform grid works best if all objects are of similar size. In the second GPU rigid body pipeline became more general, so we can deal with objects of different sizes.

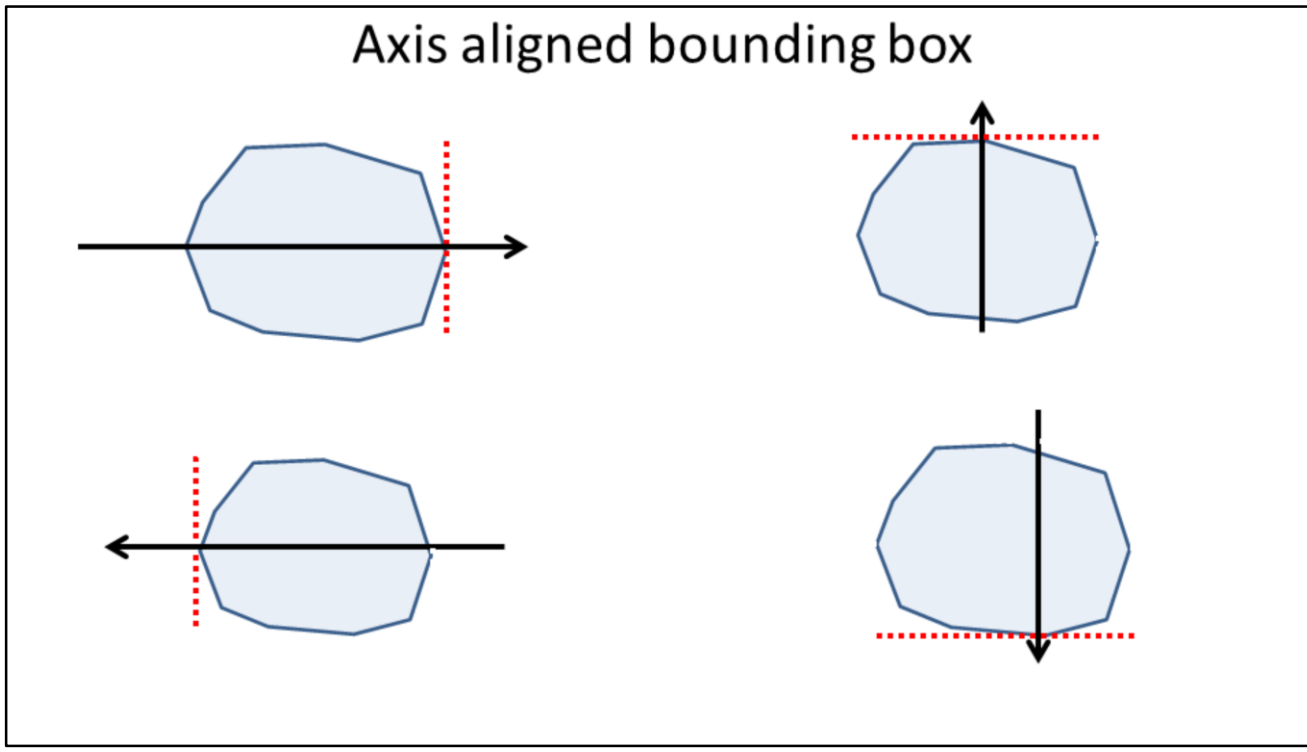
Also we use an improved collision shape representation, instead of a sphere collection, and we can use the full GPU capacity for constraint batch creation and solving.

Axis aligned bounding boxes (AABBs)



X min	X max
Y min	Y max
Z min	Z max
*	Object ID

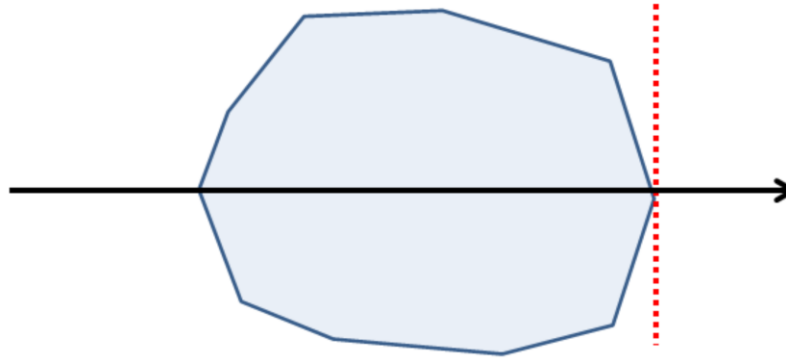
Once we use other shapes then just spheres, it becomes useful to use a simplified bounding volume for the collision shape. We use axis aligned bounding boxes on the GPU, just like we do on CPU.



We can use the support mapping function to generate local space axis aligned bounding boxes.

Support mapping

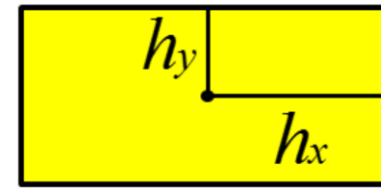
$$S_c(v) = \max\{v \cdot x : x \in C\}$$

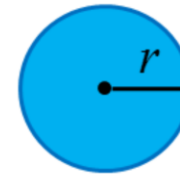


Support map for primitives

- Box with half extents h

$$S_{box}(v) = (sign(v_x)h_x, sign(v_y)h_y, sign(v_z)h_z)$$





$$S_{sphere}(v) = \frac{r}{|v|}v$$

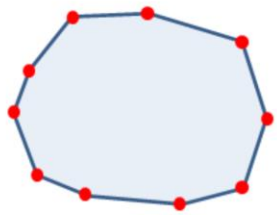
- Sphere with radius r

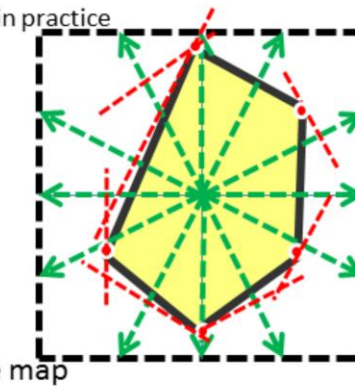
See the book "Collision Detection in Interactive 3D Environments", 2004, Gino Van Den Bergen for more information about the support mapping.

Support map for convex polyhedra

$$S_c(v) = \max\{v \cdot x : x \in C\}$$

- Brute force uniform operations (dot/max) on vertices are suitable for GPU
  - Outperforms Dobkin Kirkpatrick hierarchical optimization in practice





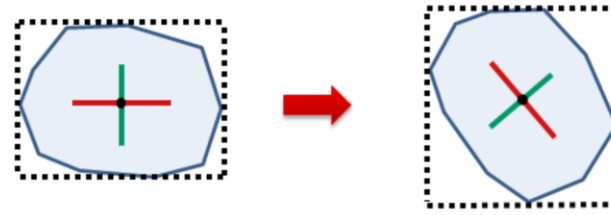
- Fast approximation using precomputed support cube map

The GPU is very suitable to compute a the dot product for many vertices in parallel. Alternatively we can use the GPU cube mapping hardware to approximage the bounding volume computation.

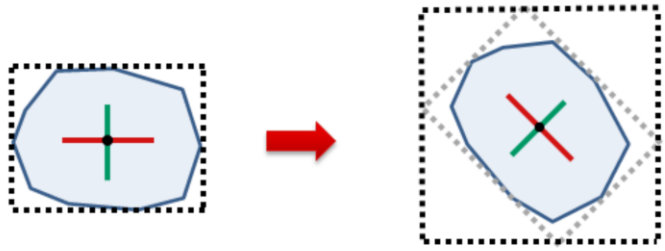
Worldspace AABB from Localspace AABB

- Affine transform

$$S_{Bx+c}(v) = B(S(B^t v)) + c$$



- Fast approximation using precomputed local aabb



- See opencl/gpu_rigidbody/kernels/updateAabbsKernel.cl

Instead of re-computing the world space bounding volume each time, we can approximate the world space bounding volume using the pre-computed local space AABB.

Host setup

```
int ciErrNum = 0;

int numObjects = fpio.m_numObjects;
int offset = fpio.m_positionOffset;

ciErrNum = clSetKernelArg(fpio.m_initializeGpuAabbsKernelFull, 0, sizeof(cl_mem), &bodies);
ciErrNum = clSetKernelArg(fpio.m_initializeGpuAabbsKernelFull, 1, sizeof(int), &numObjects);
ciErrNum = clSetKernelArg(fpio.m_initializeGpuAabbsKernelFull, 4, sizeof(cl_mem), (void*)&fpio.m_dlocalShapeAABB);
ciErrNum = clSetKernelArg(fpio.m_initializeGpuAabbsKernelFull, 5, sizeof(cl_mem), (void*)&fpio.m_dAABB);

size_t workGroupSize = 64;
size_t numWorkItems = workGroupSize*((numObjects+ (workGroupSize)) / workGroupSize);

ciErrNum = clEnqueueNDRangeKernel(fpio.m_cqCommandQue, fpio.m_initializeGpuAabbsKernel, 1, NULL, &numWorkItems,
&workGroupSize,0 ,0 ,0);
assert(ciErrNum==CL_SUCCESS);
```

AABB OpenCL™ kernel

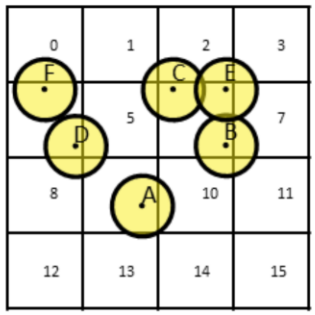
```
_void   initializeGpuAabbsFull(__global Body* gBodies, const int numNodes, __global btAABBCL* plocalShapeAABB,
__global btAABBCL* pWorldSpaceAABB)
{
      int nodeID = get_global_id(0);
      if( nodeID >= numNodes )
            return;
      float4 position = gBodies[nodeID].m_pos;
      float4 orientation = gBodies[nodeID].m_quat;
      int shapeIndex = gBodies[nodeID].m_shapeIdx;
      if (shapeIndex>=0)
      {
            btAABBCL minAabb = plocalShapeAABB[shapeIndex*2];
            btAABBCL maxAabb = plocalShapeAABB[shapeIndex*2+1];

            float4 halfExtents = ((float4)(maxAabb.fx - minAabb.fx,maxAabb.fy - minAabb.fy,maxAabb.fz -
            minAabb.fz,0.f))*0.5f;

            Matrix3x3 abs_b = qtGetRotationMatrix(orientation);
            float4 extent = (float4) (dot(abs_b.m_row[0],halfExtents),dot(abs_b.m_row[1],halfExtents),
                        dot(abs_b.m_row[2],halfExtents),0.f);
            pWorldSpaceAABB[nodeID*2] = position-extent;
            pWorldSpaceAABB[nodeID*2+1] = position+extent;
      }
}
```

See opencl/gpu_rigidbody/kernels/updateAabbsKernel.cl

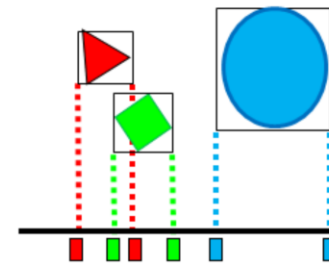
Here is the kernel code to compute the world space AABB from the local space AABB. Basically the absolute dot product using the rows of the rotation matrix and its extents will produce the world space AABB.

Note that the rows of the 3x3 orientation matrix are basically the axis of the local space AABB.

Mixed CPU/GPU pair search



	Small	Large
Small	GPU	either
Large	either	CPU

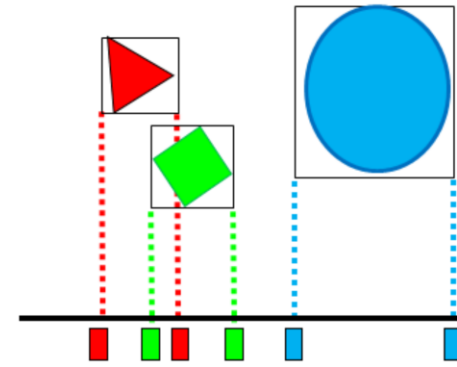


The uniform grid doesn't deal well with varying object sizes, so our first attempt to deal with mixed sized objects was to use multiple broadphase algorithms, dependent on the object size.

Computation of potential overlapping pairs between small objects can be done using a uniform grid, while a pair search that involves larger objects could be done on CPU (or GPU) using a different algorithm.

Parallel 1-axis sort and sweep

- Find best sap axis
- Sort aabbs along this axis
- For each object, find and add overlapping pairs



- Works well with varying object sizes
- See also “Real-time Collision Culling of a Million Bodies on Graphics Processing Units” http://graphics.ewha.ac.kr/gSaP

GPU SAP OpenCL™ kernel optimizations

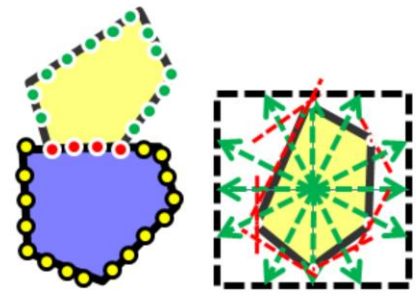
- Local memory
  - blocks to fetch AABBs and re-use them within a workgroup (requires a barrier)
- Reduce global atomic operations
  - Private memory to accumulate overlapping pairs (append buffer)
- Local atomics
  - Determine early exit condition for all work items within a workgroup
- Load balancing
  - One work item per object, multiple work items for large objects

- See opencl/gpu_broadphase/kernels/sapFast.cl and sap.cl

(contains un-optimized and optimized version of the kernel for comparison)

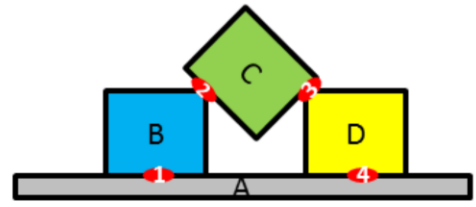
GPU Convex Heightfield contact generation

- Dual representation



- SATHE, R. 2006. Collision detection shader using cube-maps. In ShaderX5, Charles River Media

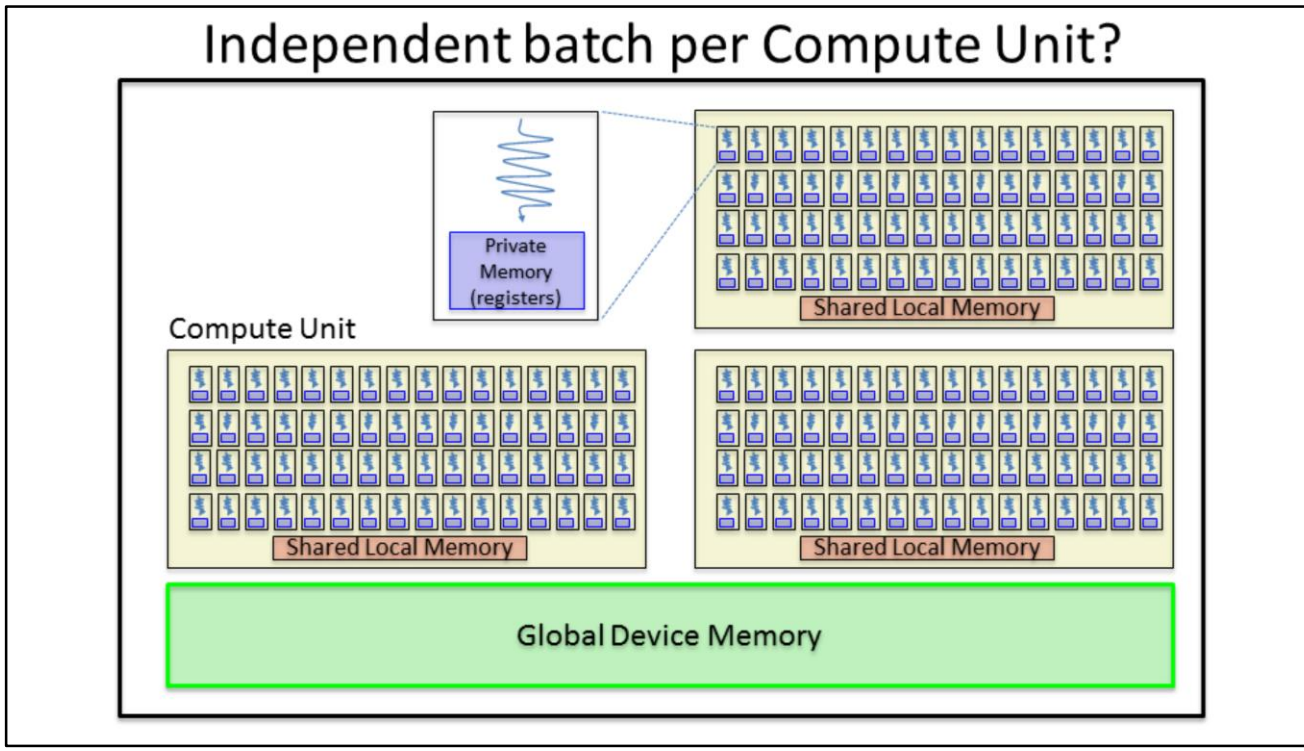
Reordering Constraints Revisited



A	B	C	D
1	1		
	2	2	
		3	3
4			4

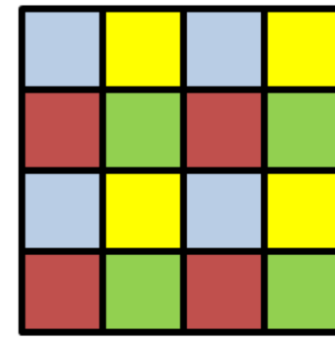
→

	A	B	C	D
Batch 0	1	1	3	3
Batch 1	4	2	2	4



If we revisit the GPU hardware, we note that Compute Units perform their work independently. So it would be good to generate batches in a way so that we can use multiple independent Compute Units.

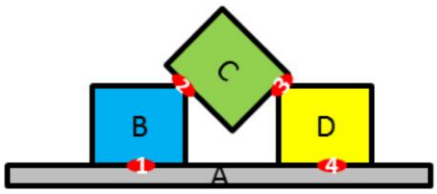
GPU parallel two stage batch creation

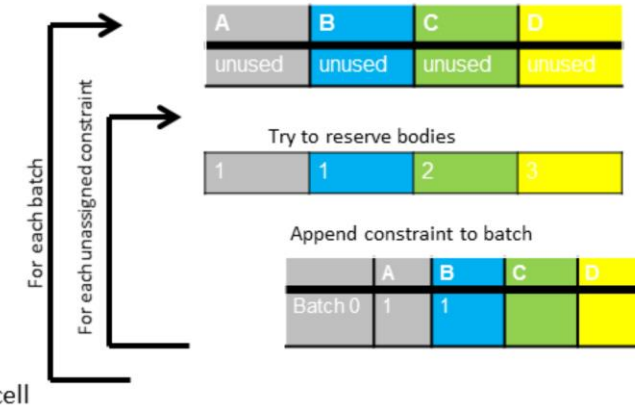


- Cell size > maximum dynamic object size
- Constraint are assigned to a cell
  - based on the center-of-mass location of the first active rigid body of the pair-wise constraint
- Non-neighboring cells can be processed in parallel

We perform the batching of constraints in two stages: the first stage splits the constraints so that they can be processed by independent Compute Units. We use a uniform grid for this, where non-neighboring cells can be processed in parallel.

GPU iterative batching





- A SIMD can process the constraints in one cell
  - cannot be trivially parallelized by 64 threads in a SIMD
- Parallel threads in workgroup (same SIMD) use local atomics to lock rigid bodies
- Before locking attempt, first check if bodies are already used in previous iterations
- See "A parallel constraint solver for a rigid body simulation", Takahiro Harada, http://dl.acm.org/citation.cfm?id=2077378.2077406

and opencl\gpu_rigidbody\kernels\batchingKernels.cl

The second stage batching within a Compute Unit is similar to the first GPU rigid body pipeline, but we added some optimizations. Takahiro Harada wrote a SIGGRAPH paper with more details.

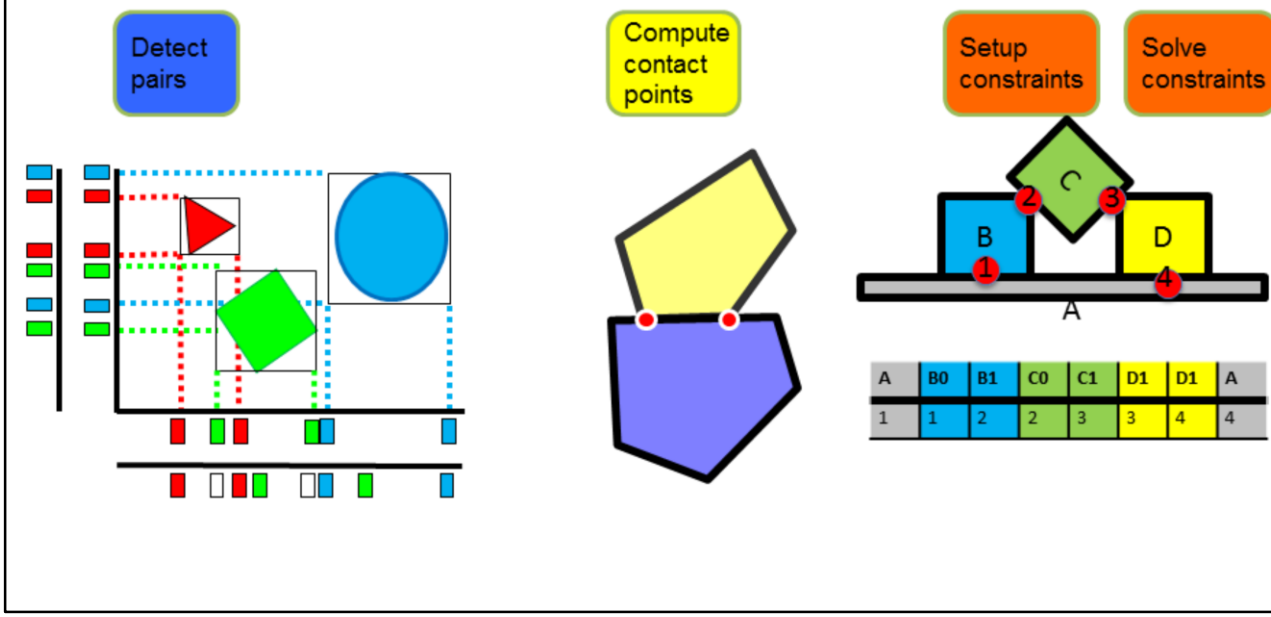
GPU parallel constraint solving

```
int idx=ldsStart+lIdx;
while (ldsCurBatch < maxBatch)   {
    for(; idx<end; )  {
        if (gConstraints[idx].m_batchIdx == ldsCurBatch)  {
            if( solveFriction )
                solveFrictionConstraint( gBodies, gShapes, &gConstraints[idx] );
            else
                solveContactConstraint( gBodies, gShapes, &gConstraints[idx] );
            idx+=64;
        } else {
            break;
        }
    }
    GROUP_LDS_BARRIER;
    if( lIdx == 0 )  {
        ldsCurBatch++;
    }
    GROUP_LDS_BARRIER;
}
```

See “A parallel constraint solver for a rigid body simulation”, Takahiro Harada, http://dl.acm.org/citation.cfm?id=2077378.2077406
Source code at opencl\gpu_rigidbody\kernels\solveContact.cl and other solve*.cl

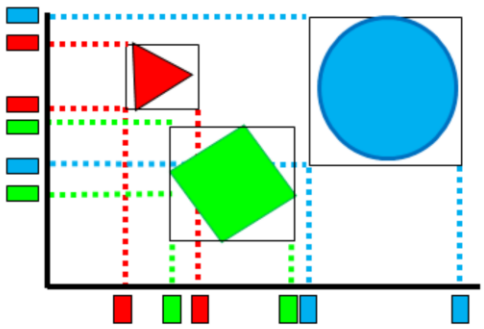
We typically use around 4 to 10 iterations in the solver, and in each iteration we have around 10 to 15 batches. The large amount of kernel launches can become a performance bottleneck, so we implemented a GPU side solution that only requires a single kernel launch per iteration.

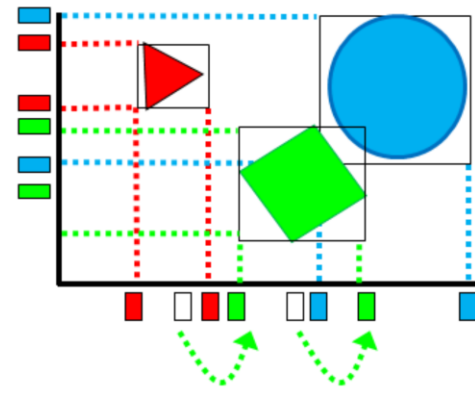
3rd GPU rigid body pipeline (2012-)



Our third generation GPU rigid body pipeline has the same quality and is as general as the regular discrete Bullet 2.x rigid body pipeline on the CPU.

Sequential Incremental 3-axis SAP



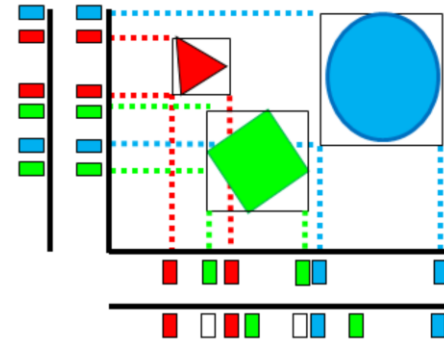


The regular 3-axis sweep and prune broadphase pair search algorithms incrementally updates the sorted AABBs for each of the 3 world axis in 3D.
We sort the begin and end points for each AABB to their new position, one object at a time, using swap operations. We incrementally add or remove pairs during those swap operations.
This exploits spatial and temporal coherency: objects don't move a lot between frames.

This process is difficult to parallelizing due to data dependencies: globally changing data structure would require locking.

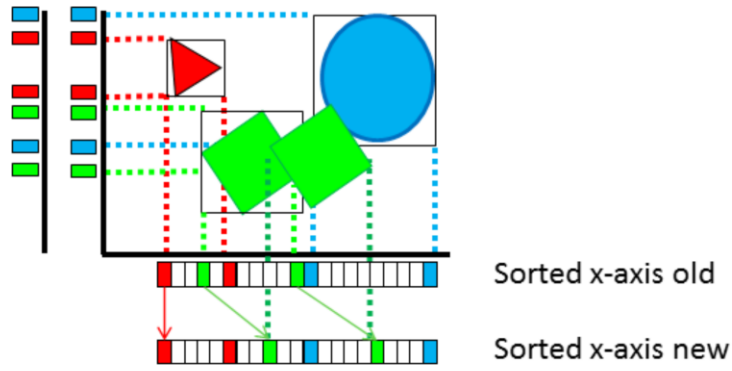
Parallel Incremental 3-axis SAP

- Parallel sort 3 axis
- Keep old and new sorted axis
  - 6 sorted axis in total



We modify the 3D axis sweep and prune algorithm to make it more suitable for GPU, while keeping the benefits of the incremental pair search during the swaps.

Parallel Incremental 3-axis SAP



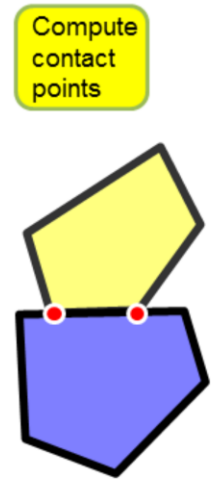
- If begin or endpoint has same index do nothing
- Otherwise, range scan on old AND new axis
  - adding or removing pairs, similar to original SAP
- Read-only scan is embarrassingly parallel

Instead of performing the incremental sort and swap together, we perform the sorting in parallel as one stage, and perform the swaps in a separate stage using read-only operations.

We maintain the previous sorted axis and compare it with the updated sorted axis. Each object can perform the swaps by traversing the previous and current sorted elements, without modifying the data.

Although unusual, we can detect rare degenerate cases that would lead to too many swaps in the 3-axis SAP algorithm, and do a fallback to another broadphase.
Such fallback is not necessary in most practical simulations. Still, generally it can be a good idea to mix multiple broadphase algorithms to exploit the best properties out of each broadphase.

Convex versus convex collision

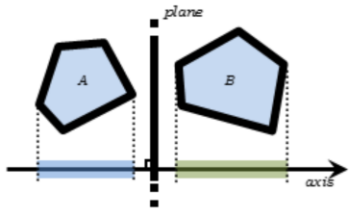


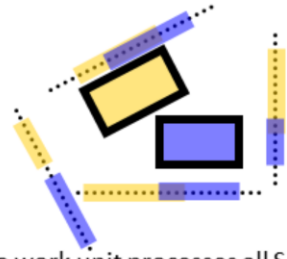
For convex objects we use the separating axis test and Sutherland Hodgeman clipping and contact reduction to generate contacts.

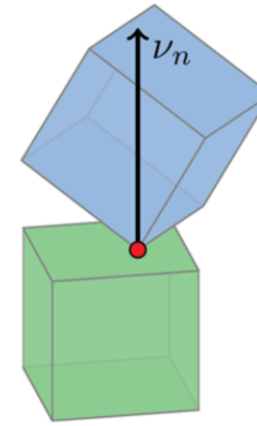
Concave compound shapes and concave triangle meshes produce pairs of convex shapes, that are processed alongside the convex-convex pairs that are produced by the broadphase.

Separating axis test

- Face normal A
- Face normal B
- Edge-edge normal



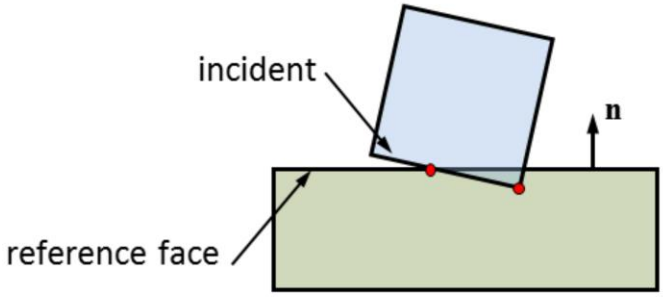


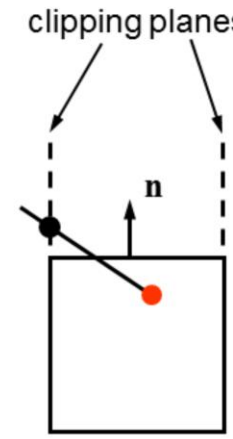


- Uniform work suits GPU very well: one work unit processes all SAT tests for o
- Precise solution and faster than height field approximation for low-resolution convex shapes
- See opencl/gpu_sat/kernels/sat.cl

Computing contact positions

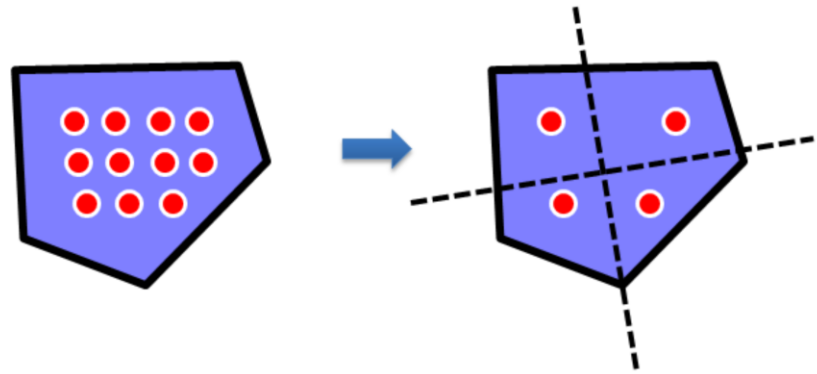
- Given the separating normal find incident face
- Clip incident face using Sutherland Hodgman clipping





- One work unit performs clipping for one pair, reduces contacts and appends to contact buffer
- See opencl/gpu_sat/kernels/satClipHullContacts.cl

GPU contact reduction



- See newContactReductionKernel in opencl/gpu_sat/kernels/satClipHullContacts.cl

The contact clipping can generate a lot of contact points. We reduce the number of contacts between two convex polyhedral to a maximum of 4 points.
We always keep the contact point with the deepest penetration, to make sure the simulation gets rid of penetrations.
We keep 3 other contact points with maximum or minimum projections along two orthogonal axis in the contact plane.

SAT pipeline

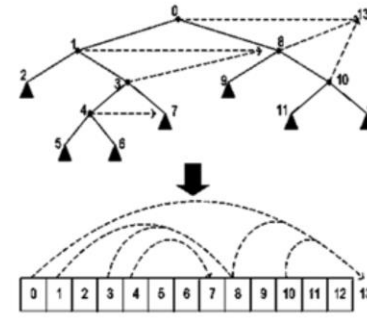
- Unified overlapping pairs
  - Broadphase Pairs
  - Compound Pairs
  - Concave triangle mesh pairs
- Break up more SAT stages to relief register pressure

The convex-convex contact generation can be used between convex objects, but also between the convex child shapes in a compound or for convex against individual triangles of a triangle mesh.
We unify the pairs generated by the broadphase pair search, compound child pair search and concave triangle mesh pairs so they can use the same narrowphase convex collision detection routines.

The code for the separating axis test, contact clipping and contact reduction is very large. This leads to very inefficient GPU usage, because of tread divergence and register spill.
We break the the convex-convex contact generation up into a pipeline that consists of many stages (kernels).

GPU BVH traversal

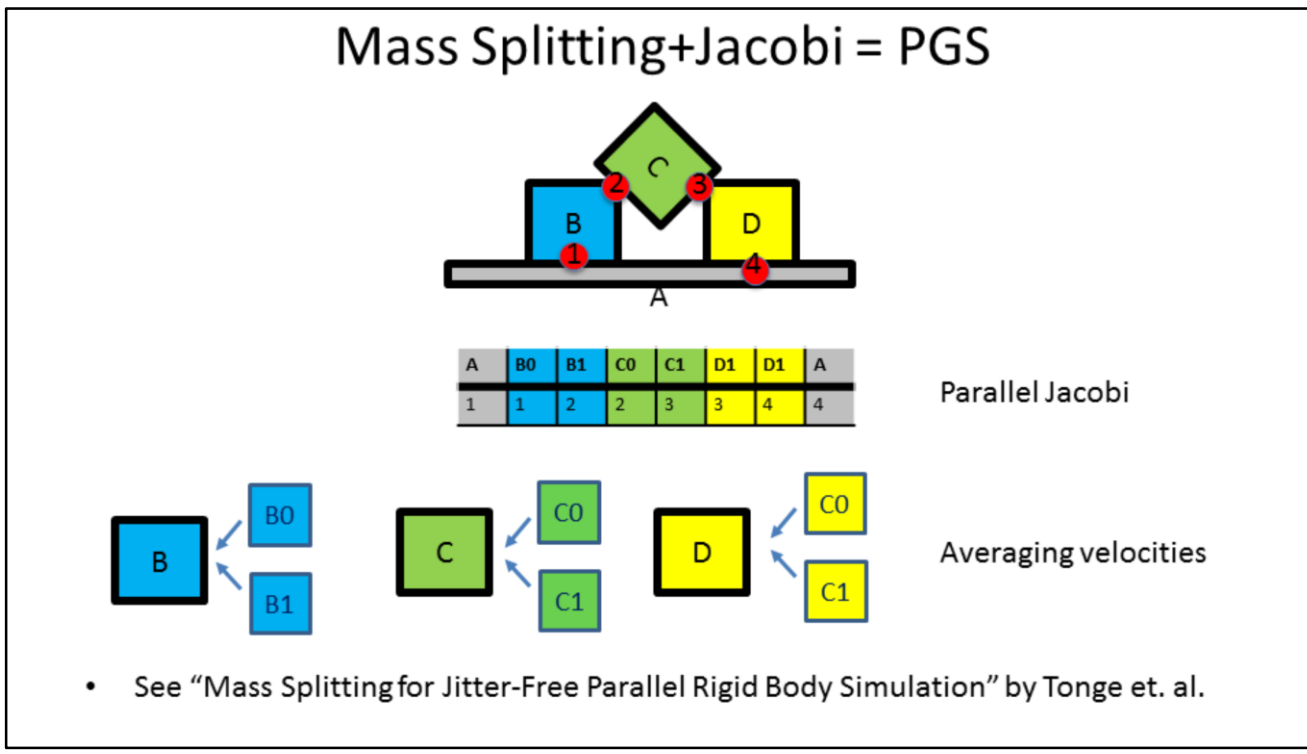
- Create skip indices for faster traversal
- Create subtrees that fit in Local Memory
- Stream subtrees for entire wavefront/warp
- Quantize Nodes
  - 16 bytes/node



When a convex or compound shape collides against a concave triangle mesh, we perform a BVH query using the world space AABB of the convex or compound shape, against the precomputed BVH tree of the triangle mesh.

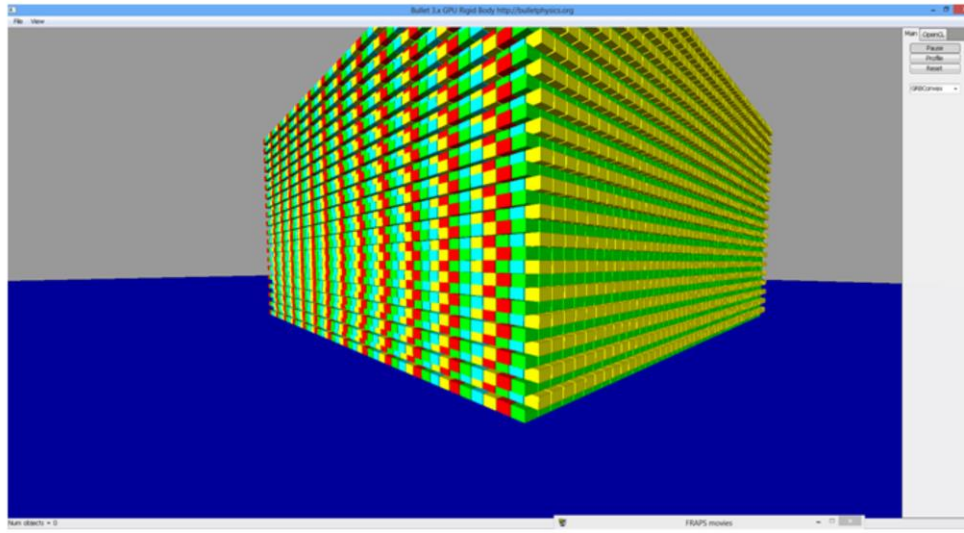
We optimized the BVH tree traversal to make it GPU friendly. Instead of recursion, we iteratively traverse over an array of nodes.

We divide the BVH tree into smaller subtrees that fit in shared local memory, so they can be processed by multiple threads in a work group.



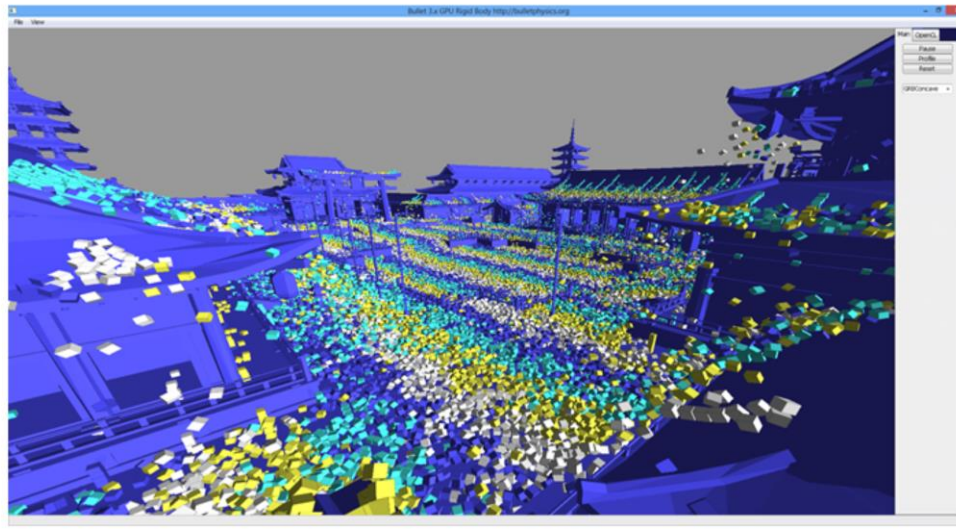
Instead of Projected Gauss Seidel, we can also use the iterative Jacobi scheme to solve the constraints.
Jacobi converges slower than PGS. We can improve the convergence by splitting the objects into multiple "split bodies" for each contact.
Then, after solving an iteration using the Jacobi solver applied to the split bodies, we average the velocities for all the split bodies and update the "original" body.
The idea is explained in the "Mass Splitting for Jitter-Free Parallel Rigid Body Simulation" SIGGRAPH 2012 paper.
We can improve upon this scheme by mixing PGS and Jacobi even further. We know that within a Compute Unit, only 64 threads are executed in parallel,
so we can only apply Jacobi within the active threads of a wavefront/warp and use PGS outside (by updating the rigid body positions using the averaged split bodies).

Test Scenario convex stack

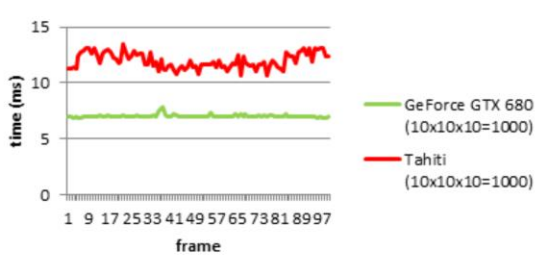


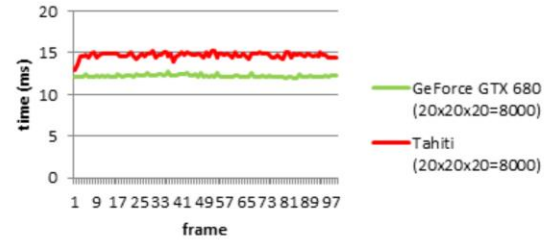
We use plain OpenGL 3.x and a simple user interface called GWEN (gui without extravagant nonsense) with some cross-platform classes to open an OpenGL 3.x context for Windows, Linux and Mac OSX.

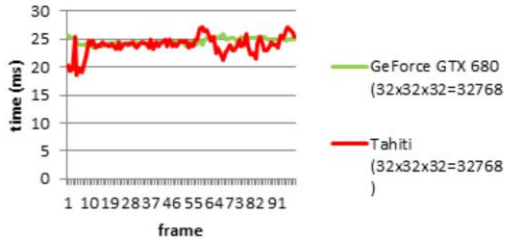
Test Scenario triangle mesh

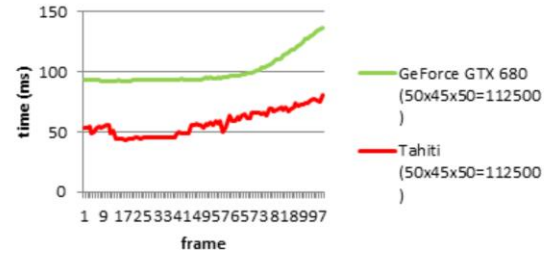


Performance









We performed some preliminary timings to make sure that our GPU rigid body pipeline works well on GPU hardware of varying vendors.

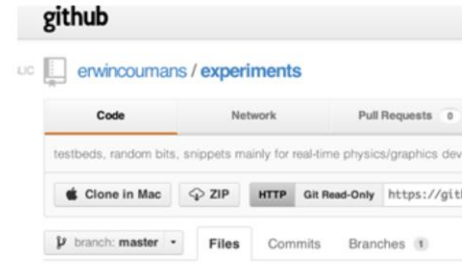
Note that those numbers don't give a good indication, because we haven't optimized the GPU rigid body pipeline good enough yet.

Timings for ½ million pairs (100k objects)

Profiling: stepSimulation (total running time: 73.233 ms) ---
0 -- GPU solveContactConstraint (45.50 %) :: 33.319 ms / frame (1 calls)
1 -- batching (13.79 %) :: 10.099 ms / frame (1 calls)
2 -- computeConvexConvexContactsGPUSAT (15.62 %) :: 11.438 ms / frame (1 calls)
3 -- GPU SAP (23.60 %) :: 17.282 ms / frame (1 calls)

Here are some preliminary performance timings. Note that we haven't optimized the GPU rigid body pipeline well, so the numbers can be improved a lot.

Build Instructions



All of the code discussed is open source

1. **Download ZIP or clone from**

http://github.com/erwincoumans/experiments

Windows Visual Studio

2. **Click** on build/vs2010.bat
3. **Open** build/vs2010/0MySolution.sln

Mac OSX Xcode or make

2. **Click** on build/xcode.command
3. **Open** build/xcode4/0MySolution.xcworkspace

We test our GPU rigid body pipeline on Windows, Mac OSX and Linux using AMD, NVIDIA and Intel GPUs. The source code will be available at the github repository.

Thank You!

- You can visit the forums at http://bulletphysics.org for further discussion or questions
- See previous slide for source code instructions