Cell Based Servers for Next-Gen Games

Cell BE Online Game Prototype

Bruce D’Amora
T.J. Watson Research
Yorktown Heights, NY

Karen Magerlein
T.J. Watson Research
Yorktown Heights, NY

| 19 June 2007 |
Agenda

- Motivation for Research
- Flow & Execution of Prototypical Online Game
- Operation of Physics Engine
- Bandwidth Management
- Performance
- Future Work
Motivation: Server-side Physically Based Modeling

- Enable the next generation MMOGs & virtual environments
  - Current online video games perform limited amount of physical simulation
    - Not enough client CPU resources
    - Bandwidth & Latency between processing nodes prohibitive to achieving real time performance
- Enable complex visual F/X movies on large servers
Game Execution Flow

Server

- Entity State Verification Input from Client
- Network Scene Management
- Entity State Input from Server
- Player Input

Client

- Server Model of World State
- Simulation
- Player State Input from Client
- Client Model of World State
- Rendering
- Players View

Execution & Data Flow
Server-side state: Stateless model

Compute Nodes

Node 0
- Blade 0
  - Network Compression
  - Arithmetic Coding
  - Rigid Body
  - Collisions

Node 1
- Blade 1
  - Network Compression
  - Arithmetic Coding
  - Rigid Body
  - Collisions

Node n
- Blade n
  - Network Compression
  - Arithmetic Coding
  - Rigid Body
  - Collisions

Virtual World
- State Manager

Game Framework
- Player Login
- Event Handlers

Network Manager

Dispatcher Node

Client
- Client Renderer
- Interpolator
- Local Database
- Event Handler

Dispatch work (elements of Virtual World) to Helper Nodes
## Bandwidth Requirements

**Uncompressed data transmitted to compute notes:**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Size (in 32-bit floats)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>3</td>
</tr>
<tr>
<td>Orientation</td>
<td>4</td>
</tr>
<tr>
<td>Linear Velocity</td>
<td>3</td>
</tr>
<tr>
<td>Angular Velocity</td>
<td>3</td>
</tr>
<tr>
<td>Mass</td>
<td>1</td>
</tr>
<tr>
<td>Inertia Tensor</td>
<td>6</td>
</tr>
<tr>
<td>Coefficient of Friction</td>
<td>1</td>
</tr>
<tr>
<td>Collision Flags</td>
<td>1</td>
</tr>
<tr>
<td>Collision Body</td>
<td>(unbounded)</td>
</tr>
</tbody>
</table>

**Only variable data transmitted back to dispatcher:**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Size (in 32-bit floats)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>3</td>
</tr>
<tr>
<td>Orientation</td>
<td>4</td>
</tr>
<tr>
<td>Linear Velocity</td>
<td>3</td>
</tr>
<tr>
<td>Angular Velocity</td>
<td>3</td>
</tr>
</tbody>
</table>

Reduce the bandwidth requirements by using a dictionary index and some scaling and options information for each body.
Collision Detection

- Broad-phase collision detection
  - simple bounding box culling
- Mid-phase collision detection consists of a 1D sort and sweep of the axis-aligned bounding boxes
- Narrow-phase collision detection uses a proprietary algorithm to compute a set of contacts between a pair of convex hulls

- All collision detection being moved to SPEs
Server Side Physics Loop

Runtime steps:

1. **Collision detection** (broad-, mid-, and narrow-phase)
2. **Revise lists** of active vs. sleeping bodies
3. **Partition active bodies** into non-interacting groups
4. **Decouple** to break up groups which are too large to fit in SPE local store
5. **Integrate** to obtain positions of bodies for next step
6. **Adjust step size** if integration step fails
Rigid Body Dynamics

- Objects in the game world are represented by one or more rigid bodies;
  - a sparsely populated world will have about 1000 rigid bodies
    - 6 degrees of freedom per rigid body
  - Represented by 3-tuples, but stored as padded vector floats:
    - Linear position of the body’s center of mass
    - Linear velocity of body
    - Angular velocity of body
    - Orientation representation is a unit quaternion or matrix $3 \times 3$
- Forces and constraints define interactions between rigid bodies
  - allow joints, hinges, etc. to be implemented
- Physics provides real-time simulation of the interaction between rigid bodies
Sparse Matrix Data Structures on Cell

- Matrix is block-sparse with 6x6 blocks
  - diagonal blocks represent bodies and
  - off-diagonal blocks represent forces between bodies
- Typical 65-body group has ~200 nonzero blocks in a 65x65-block matrix
- Diagonal elements are assumed nonzero and are stored as a “block” vector for fast access
- Off-diagonal elements are stored in linked lists (one per block row) of block data and associated block column position
- 6x6 float block data is currently stored in column-major form in a padded 8x6 block for ease of access
- Vectors used in sparse matrix multiplication are similarly stored as vector floats on SPE with one unused float per three elements
Numerical Integration

- Game world is partitioned into non-interacting groups of 1 or more rigid bodies which can be simulated on a single SPU
  - maximum of about 120 bodies per group
- SPU performs semi-implicit integration step for a second-order rigid body dynamics system
  - uses conjugate gradient squared algorithm
  - basic operation is multiplication of a 6x6-block-sparse matrix by vector
- Output of the integration step
  - change in linear velocity and angular velocity for each rigid body over one time step
SPU Implementation: Rigid Body Structures

**Input structure**

```c
struct Rigid_Body {
    //state
    Vec3 position;
    Matrix33 orientation;
    Vec3 velocity;
    Vec3 angular_velocity
    //mass params
    float inverse_mass;
    Matrix33 inverse_inertia;
    //other params:
    float coeffecient_friction;
    float coeffecient_damping;
    ...
} bodies[num_bodies];
```

**Output structure**

```c
struct Rigid_Body_Step {
    Vec3 delta_velocity;
    Vec3 delta_angular_velocity;
} delta_v[num_bodies];
```

Forces can be global, unary, or binary

- Examples of binary forces:

```c
struct Point_To_Point_Constraint_Force {
    int index_body_a;
    int index_body_b;
    Vec3 point_body_space_a;
    Vec3 point_body_space_b;
};
struct Contact_Force {
    int index_body_a;
    int index_body_b;
    Vec3 point_world_space;
    Vec3 normal_world_space;
    float penetration;
};
```
Intermediate data structures

Vec4 v0[2*num_bodies];
Vec4 f0[2*num_bodies];
- Six component vectors are padded out to 8 components, with one float of
  padding on each of the linear and angular components
- The most complicated data structure is the block sparse matrix:
  struct Block_Sparse_Matrix {
    struct Block {
      Matrix86 m;
      int column_index;
      Element* pointer_next;
    };
    Block* rows[NUM_BODIES];
  };
- The logically 6x6 blocks are padded to 8x6. The matrix is stored in a
  column major fashion, with padding on the 4th and 8th element to match
  padding in v0 and f0
Numerical Integration Steps

Steps 1-4 are performed on the SPE.

1. Calculate the components of $A$ and $b$. $v_0$ and $W$ are trivial to extract. $f_0$ must be calculated. $df_{dx}$ and $df_{dv}$ both require considerable computational effort to calculate.

2. Form $A$ and $b$

3. solve $A*delta_v = b$ by a conjugate gradient method.

4. step the system from $Y_0$ to $Y_1$ by $delta_v$
Conjugate Gradient Squared

Why was conjugate gradient squared chosen?
- The preferred choice is bi-conjugate gradient, but this requires multiplies by $A$ transpose
- The sparse matrix transpose times vector can be written in a row-oriented fashion, but having the inner 6x6 logical block efficiently support both multiplication with a logical 6-vector and multiplication of its transpose with a logical 6-vector may be more expensive than the alternative – conjugate gradient squared.
  - Caching the transpose of the blocks would likely take too much memory
SPU Conjugate Gradient Squared

```c
int SPU_conjugate_gradient_squared (  
    int* Pdest_num_iters,  
    vf3 x[],  
    const SPU_Sparse_Matrix_Element* const A[],  
    const vf3 pb[],  
    int num_bodies)
{
    nv = 2*num_bodies;
    float rho_1=0.0f, rho_2=0.0f, alpha=0.0f,  
    beta=0.0f;

    int iter = 0;
    float resid = 0.0f;

    vf3* b = (vf3*)pb;

    const int max_iters = 2*6*num_bodies;
    //
    float bb = vec_dot(b,b);
    vector float bbb = spu_splats(bb);
    bbb = _sqrtf4(bbb);
    const float norm_b = spu_extract(bbb, 0);
    if (norm_b < .0001f)
        return 0;

    const float rel_tol = 0.001f;
    const float abs_tol = 0.001f;
    neg_mul_add(r, A, x, b);
    vec_assign(r_tilde, r);

    while (1) {
        float rr = vec_dot(r, r);vector float rrr = spu_splats(rr);
        rrr = _sqrtf4(rrr);
        rr = spu_extract(rrr, 0);
        resid = rr/norm_b;
```
if ((resid < rel_tol) || (rr < abs_tol)) {
    return 0;
} else if (iter >= max_iters) {
    return 1;
} ++Pdest_num_iters;

rho_1 = vec_dot(r_tilde, r);
if (rho_1 == 0.0f) {
    return 2;
} if (iter == 0) {
    vec_assign(u, r); vec_assign(p, u);
} else {
    beta = rho_1/rho_2;
    vec_add_scaled(u, r, beta, q);
    vec_add_scaled(p, q, beta, p);
    vec_add_scaled(p, u, beta, p);
}

vec_assign(p_hat, p);  // p_hat = precond.solve(p);
mul(v_hat, A, p_hat);
float den = vec_dot(r_tilde, v_hat);
if (den == 0.0f) {
    return 3;
} alpha = rho_1/den;

vec_add_scaled(q, u, -alpha, v_hat);
vec_add(u, u, q);
vec_assign(u_hat, u);  // u_hat = precond.solve(u);
vec_add_scaled(x, x, alpha, u_hat);
mul(q_hat, A, u_hat);
vec_add_scaled(r, r, -alpha, q_hat);

rho_2 = rho_1;
++iter;
} // end while
return 0;
} // end of spu_conjugate_gradient_squared
SPU Conjugate Gradient Squared Method

- The conjugate gradient squared method only requires \( A \) times a vector – however, it has been found in practice to converge more slowly.
- Each iteration of the conjugate gradient performs two matrix vector products along with a handful of vector scales, adds, and inner products. The matrix product is the only non-trivial operation. It looks like this:

```c
void mul(Vec8* res, const Block_Sparse_Matrix2& A, const Vec8* x)
{
    for (int i = 0; i < num_bodies; ++i) {
        Vec8 sum = 0;
        for (Block* b=A.rows[i]; b; b = b->pointer_next)
            sum += b->m * x[b->column_index];
        res[i] = sum;
    }
}
```

Where \( b->m * x[b->column_index] \) is pseudo code for Column_Major_Matrix86 times Vec8 which is basically trivial SPE code.
SPU Sparse Matrix Multiply Code

```c
void mul(vf4 d[], const SPU_Sparse_Matrix_Element* const A[], const vf4 x[])
{
    PROFILER(mul);
    int i;
    for (i=0; i < nv/2; ++i) {
        const SPU_Sparse_Matrix_Element* p = A[i];

        vf3 s0 = vf3_zero;
        vf3 s1 = vf3_zero;

        while (p) {
            int j = p->j;
            s0 = spu_add(s0, xform_vf3(&p->a.a[0][0], x[2*j+0]));
            s0 = spu_add(s0, xform_vf3(&p->a.a[0][1], x[2*j+1]));
            s1 = spu_add(s1, xform_vf3(&p->a.a[1][0], x[2*j+0]));
            s1 = spu_add(s1, xform_vf3(&p->a.a[1][1], x[2*j+1]));

            p = p->Pnext;
        }
        d[2*i+0] = s0;
        d[2*i+1] = s1;
    }
}
```
Memory constraints and workload size

- The number of matrix blocks required is less than num_bodies + 2*num_binary_forces
- A typical 65 rigid body scene had approximately 400 contacts
- SPU memory usage for integrating this example scene follows:
  
  **Input:**
  \[
  \text{num\_bodies}\times\text{sizeof}(\text{Padded(Rigid\_Body)}) = 65\times160\text{B} = 10400\text{B} \\
  \text{num\_contacts}\times\text{sizeof}(\text{Padded(Contact\_Force)}) = 400\times48\text{B} = 19200\text{B} \\
  \text{TOTAL} = 29600\text{B}
  \]

  **Output:**
  \[
  \text{num\_bodies}\times\text{sizeof}(\text{Padded(Rigid\_Body\_Step)}) = 65\times32\text{B} = 2080\text{B}
  \]

  **Intermediate:**
  \[
  \text{num\_bodies}\times\text{sizeof}(\text{Padded(W\_Element)}) = 65\times64\text{B} = 4160\text{B} \\
  \text{num\_vectors}\times\text{num\_bodies}\times\text{sizeof}(\text{Padded(Vec6)}) = 8\times65\times32\text{B} = 16640\text{B} \\
  \text{num\_bodies}\times\text{sizeof}(\text{Block*}) = 65\times4\text{B} = 260\text{B} \\
  \text{num\_blocks}\times\text{sizeof}(\text{Padded(Block)}) = 200\times208\text{B} = 41600\text{B} \\
  \text{TOTAL} = 62660\text{B}
  \]

- Including double buffering the input and output areas, we use a total of 126,020B
- Maximum workload is probably less than 120 bodies
Integration Core Performance

3.2 Ghz P4/SSE versus 2.4 Ghz Cell BE
1 SPE about 1.5x P4/SSE
Relative Code Segment Performance

<table>
<thead>
<tr>
<th>Time (secs)</th>
<th>P 4</th>
<th>Cell BE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Integration**
- **Collision Detection**
- **Grouping**
- **Decoupling**
Cell BE Code Sizes for Numerical Integration

- Currently 7,104 lines of SPE code
  - 2,741 lines of human written code
  - 4,363 lines of generated code
- 4,536 lines of PPE support code
  - Decouple
  - Pack
  - Unpack
  - Schedule
- 828 lines of PPE library code
  - DMA commands wrapper/emulation
  - Scalar/SSE/VMX emulation of SPE ops
Conclusions & Next Steps

- SPEs are very fast and can provide significant speedup
- DMA engine provides low-latency, high bandwidth data movement
- PPE should be used as a control processor
  Maximum PPE performance achieved by using 2 PPE threads
- **Move as much of code to SPEs – even scalar code**
  - Collision detection being moved to SPEs
  - Ultimately grouping, de-coupling and network scene management should be moved to SPEs.
- Data structures should be optimized for transfer to SPE LS
  - Lot of PPE cycles wasted in packing and unpacking data
- Add more destructible-body and soft-body dynamics
Thank You!
Acknowledgements

- Randy Moulie
- Sidney Manning
- Bob Hanson
- Jeff Derby
- Ashwini Nanda