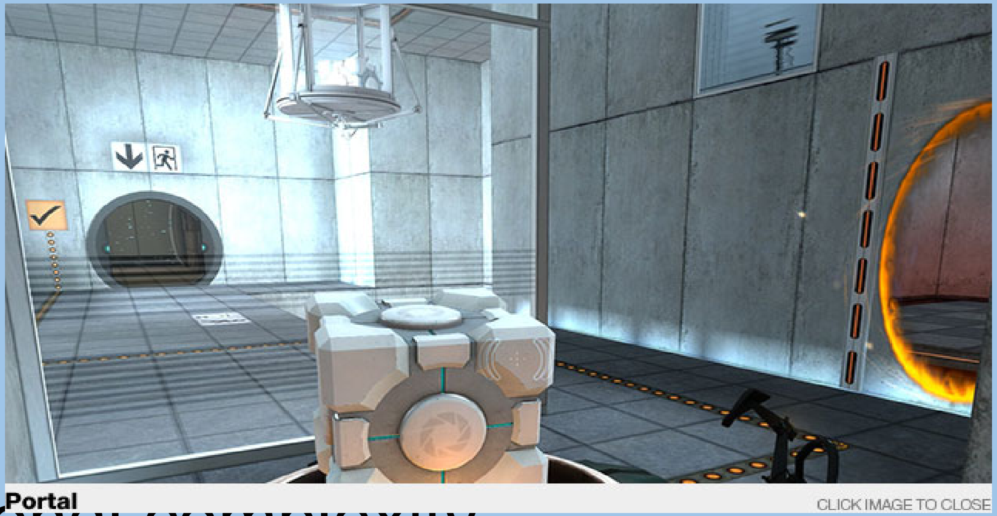
GPGPU Accelerated 2D Rigid Body Physics Engine

CPE790 Final Project Presentation
6/2/11

Falco Girgis

Motivation

- Physics simulations are becoming increasingly popular in video games



- High computational complexity
- Modern GPUs are optimized for vector/matrix floating point calculations required for rendering
- Shaders introduce programmable GPU
- Physics calculations - series of independent, intensive floating point calculations performed on each object in a scene
- Inherent potential to exploit data-level parallelism!

Project Goals

- Create 2D rigid body dynamics engine
- Implement the engine fully in software (C language)
- Port parallelizable portions to the GPU using OpenCL for hardware acceleration
- Analyze performance trade-offs, advantages, or disadvantages
- Determine which portions of a physics engine stand to benefit most from SIMD hardware acceleration

Outline

1. Data structures
2. Math/Linear algebra routines
3. Force Generators
4. Physics Pipeline
    1. Integration
    2. Transformation
    3. Collision Detection/Contact Generation
    4. Contact Resolution
        1. Velocity
        2. Interpenetration
5. OpenCL Acceleration
    1. Memory management
    2. Pipeline Kernels
    3. Performance
    4. Additional Considerations/Future Work

Data Structures

- Statically allocated arrays serving as data "pools" for all object types
- Objects index into each other's pools

```
typedef struct _gyroCollidable {
    gyroInstanceTransform instMat;
    gyroVector2 worldVertex[4];
    gyroVector2 axes[2];
    GYuint32 partIndex;
    GYuint16 layer, layerAgainst;
    char type;
} gyroCollidable;
```

```
typedef struct _gyroParticle {
    gyroVector2 _velocity;
    gyroVector2 _acceleration;
    gyroVector2 _prevAccel;
    gyroVector2 _forceAccumulator;
    float _rotation;
    float _angularAccel;
    float _torqueAccum;
    unsigned int _collidableIndex;
    unsigned int _attribIndex;
}  gyroParticle;
```

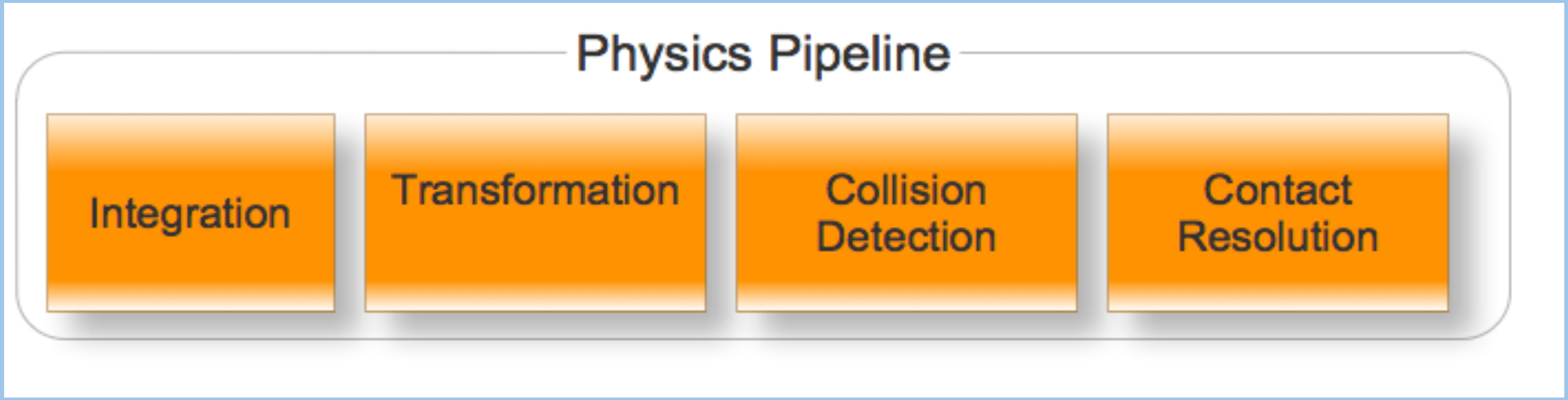
```
typedef struct _gyroStaticAttrib {
    float _inverseMass;
    float _coeffOfRestitution;
    float _damping;
    float _invMomOfInertia;
    float _angDamping;
}  gyroStaticAttrib;
```

Vector/Matrix Math

- Underlying mechanism abstracted away from physics calculations
- Easily take advantage of SIMD/OpenCL functions later

```
void gyroVector2Perp(gyroVector2 *const dest, const gyroVector2 *const src);

void gyroVector2Scale(gyroVector2 *const dest, const gyroVector2 *const vec, const float scalar);

void gyroVector2ScaleTo(gyroVector2 *const dest, const float scalar);

void gyroVector2VectorCrossProd(gyroVector2 *const dest, const gyroVector2 *const vec, const float scalar);

float gyroVector2ScalarCrossProd(const gyroVector2 *const src1, const gyroVector2 *const src2);

void gyroVector2Add(gyroVector2 *const dest, const gyroVector2 *const src1, const gyroVector2 *const src2);

void gyroVector2AddTo(gyroVector2 *const dest, const gyroVector2 *const src);
void gyroVector2RotateOrigin(gyroVector2 *const dest, const gyroVector2 *const src, const float angle);
```

Physics Pipeline



Integration

- Newtonian physics
- Function of "deltaTime" -- time elapsed since the previous frame

1. Converts external forces into linear acceleration and torque applied to an object
2. Updates linear velocity from acceleration and angular velocity from torque
3. Damps both linear and angular velocity (to simulate friction/velocity loss)
4. Updates position and orientation from linear velocity and angular velocity
5. Clears force and torque accumulators for the next frame

Transformation

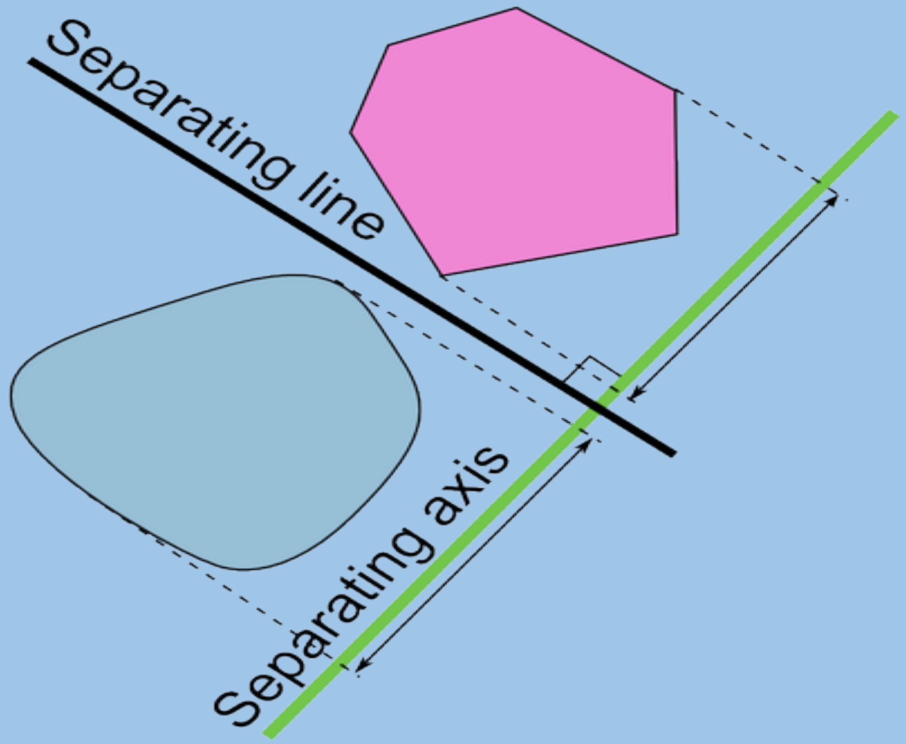
- OpenGL needs each object's vertices in world coordinates for rendering
- Collision detection algorithms need each object's vertices in world coordinates for contact generation

1. Create an instance matrix based on the state of the object (size, orientation, position)
2. Transform the object's local vertices by this matrix and store the world coordinates
3. Create two vectors forming the perpendicular axes of the quads and normalize them for later use (collision detection).

Collision

1. Broad Phase
   - Reduce amount of collision checks from every object against every object to avoid unnecessary collision checks
   - Reduce time complexity of collision detection from O (n^2) to O(logn)
   - Not implemented
2. Narrow Phase
   - Actual collision check against object geometry
   - Provides data pertaining to colliding bodies to physics pipeline
   - Implemented using Separating Axis Theorem

Separating Axis Collision Detection



1. Project each polygon edge onto each of the two body's normalized axes
2. If there is an overlap on each axis, the axis of minimal overlap is the separating axis (contact normal)
3. If ANY axis does not overlap, no collision has occurred

Contact Generation

1. □Contact Normal
    - Direction from which the collision occurred
1. Penetration Depth
    - Scalar representing the magnitude of interpenetration
2. Two bodies involved
3. Point of Contact
    - Currently VERY rough approximation
4. Coefficient of Restitution
    - Amount of momentum conserved from collision (dependent upon materials)
5. Separating Velocity
    - Speed at which two objects are separating in the direction of the collision normal
6. Total Inertia
    - Sum of inertia from both bodies due to linear and angular quantities

Contact Resolution

- Various approaches
- Chose Impulse-based approach
  - Iterative
  - Allows objects to interpenetrate
  - Models collisions as individual, instantaneous changes in velocity and position, rather than a sum of all forces
  - Faster, but less mathematically accurate
  - Two independent steps
    1. Velocity Resolution
    2. Interpenetration Resolution

Velocity Resolution

- Separating velocity
  - rate at which the two objects are separating in the direction of the contact normal
  - DotProd(relativeVelocity, normal);
- Desired change in velocity
  - according to the conservation of momentum (separating velocity and coefficient of restitution)
- Calculate total inertia in the direction of the contact normal
- Calculate an "impulse" that will result in the desired change in velocity
- Apply this impulse to 1) body A's linear velocity 2) body A's angular velocity 3) body B's linear velocity 4) body B's angular velocity in proportion to their contributions to the overall inertia

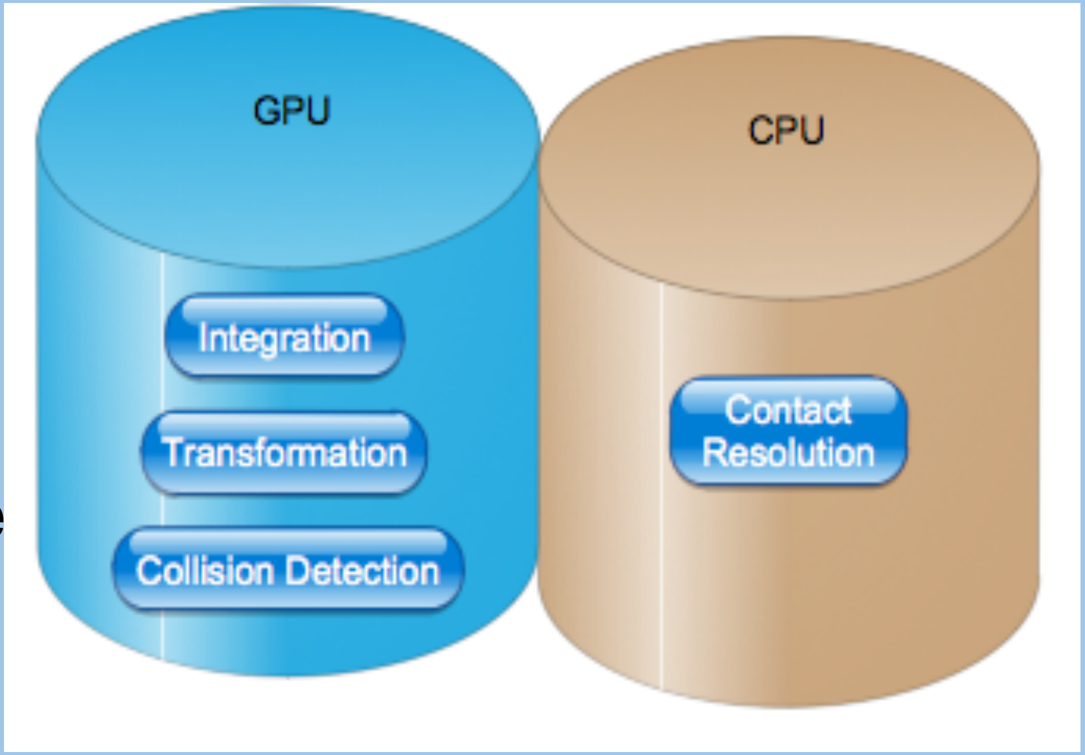
Interpenetration Resolution

- "Nonlinear Projection"
- Essentially the same as velocity resolution
- Consider penetration depth in the direction of the contact normal the overall desired change in position (not velocity)
- Perform inertial calculations in the direction of the contact normal
- Apply a proportion of the total impulse to 1) body A's position 2) body A's orientation 3) body B's position 4) boy A's orientation in proportion to their contributions to the overall inertia

Migrating to the GPU

- New pipeline:
  - GPU-side
    1. Integration
    2. Transformation
    3. Collision Detection
  - CPU-side
    1. Contact Resolution

1. Data Alignment
2. Memory Buffers/Manageme
3. Auxiliary Functions
4. Kernel Implementations



Structure Data Alignment

- Compilers introduce different amounts of padding into structures for optimization
- This padding and the size of each structure must remain consistent from the CPU to the GPU
  - __attribute__ ((aligned (16)))

```
typedef struct _gyroParticle {
    gyroVector2 _velocity;
    gyroVector2 _acceleration;
    gyroVector2 _prevAccel;
    gyroVector2 _forceAccumulator;
    float _rotation;
    float _angularAccel;
    float _torqueAccum;
    unsigned int _collidableIndex;
    unsigned int _attribIndex;
} __attribute__ ((aligned (16))) gyroParticle;
```

Memory and Buffer Allocation

- Attribute Pool - Read-Only global memory
- Collidable Pool - Read/Write global memory
- Particle Pool - Read/Write global memory
- Written to/read from GPU once-per-frame

```
err = clEnqueueWriteBuffer(_commandQueue, _attribMem, CL_TRUE, 0, sizeof
(gyroStaticAttrib) * _attribCount, _attribPool, 0, NULL, NULL);

err |= clEnqueueWriteBuffer(_commandQueue, _collidableMem, CL_TRUE, 0, sizeof
(gyroCollidable) * _colCount, _colPool, 0, NULL, NULL);

err |= clEnqueueWriteBuffer(_commandQueue, _rigidBodyMem, CL_TRUE, 0, sizeof
(gyroParticle) * _partCount, _partPool, 0, NULL, NULL);
```

Integration Kernel

- Fairly straightforward
- Requires fetching of both attribute and collidable objects corresponding to each rigid body/particle
- Benefits from fast floating-point calculations
  - powf()
  - basic vector functions

Transformation Kernel

- Calculates world coordinates for each Collidable
- Normalizes two axes for each Collidable for the OBBCollision Kernel
- Benefits from:
    - sin/cos lookups
    - vector/matrix operations
    - inverse sqrt() approximation

OBB Collision Kernel

- Hardest to implement, by far
- Required a GPU-side "contact pool" for generated contacts
- How to regulate access to this pool?
    - "kind of standard" atomic functions provided by some versions of OpenCL:

```
#pragma OPENCL EXTENSION cl_khr_global_int32_base_atomics : enable
#pragma OPENCL EXTENSION cl_khr_local_int32_base_atomics : enable
#pragma OPENCL EXTENSION cl_khr_global_int32_extended_atomics : enable
#pragma OPENCL EXTENSION cl_khr_local_int32_extended_atomics : enable
```

For my savior:

```
unsigned int colIndex = atom_inc(conCount);
__global gyroContact *contact = &conPool[colIndex];
```

Contact Resolution

- Not parallelizable
- Each contact that is resolved can effect other contacts
- Requires retraversing contact list after each contact is updated
- Quickly becomes program bottleneck

Overall Performance Observations

1. Initial overhead of allocating buffers and transferring data to GPU is fairly large.
2. Kernels all benefitted most from parallel execution as opposed to speed-up from OpenCL math routines
3. Magical "break-even" point is when the latency of CPU-based calculations becomes greater then the PCI-E BUS latency
4. Kernel speed from fastest to slowest
    1. Integration
    2. Transformation
    3. OBBCollision
5. Bottleneck becomes CPU-based contact resolution

Additional Optimizations

1. Better use of built-in vector ops
2. more consideration for cache-friendly alignments (card specific)
3. OpenGL Interoperability
4. Additional resting contact code
5. Broad-phase collision detection
6. Contact grouping
7. Parallelizable contact resolution algorithm?

Future Work

1. Other convex shapes
2. CORRECT Point of Contact
3. Hardware accelerated on the CPU with SIMD/SSE vector/matrix instructions
4. complex forces: springs, ropes, joints

Works Cited

1. Ian Millington, "Game Physics Engine Development"
2. Danny Kodicek, "Mathematics and Physics for Programmers"
3. Khronos Group, "OpenCL Quick Reference Guide"
4. Aaftab Munshi; Benedict Gaster; Timothy G. Mattson; James Fung; Dan Ginsburg, "OpenCL Programming Guide (Rough Cuts)"
5. Khronos Group, http://www.khronos.org/registry/cl/
6. NVidia OpenCL Programming Guide: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/OpenCL_Programming_Guide.pdf
7. Richard S. Write, Jr.; Benjamin Lipchak; Nicholas Haemel, "OpenGL SuperBible"