Deformable Body Simulation on GPU

Nuttapong Chentanez

Why deformable bodies?

- Looks more real than rigid bodies
  - Most objects in the real world deform, true rigid bodies don't physically exist
- Open up new possibilities in gaming experiences
- GeForce 8800 can handle the computations necessary for deformable body simulation entirely on the GPU
  - Simulation
  - Collision detection and response
  - Rendering

Previous works on "Real Time" simulation of deformable bodies

- Physically based
  - From Solid Mechanics
    - Start from Stress-Strain relationship
    - Derive governing Partial Differential Equation (PDE)
    - Discretize to ODE and Solve
      - Explicit Integration – Unstable for reasonable time step
      - Implicit Integration – More complex to implement
    - May perform dimension reduction to reduce run-time complexity
      - Very long pre-processing time
  - Examples
    - Modal Analysis [1]
    - Interactive Virtual Materials [2]
    - Reduced nonlinear model [3]

Previous works on "Real Time" simulation of deformable bodies

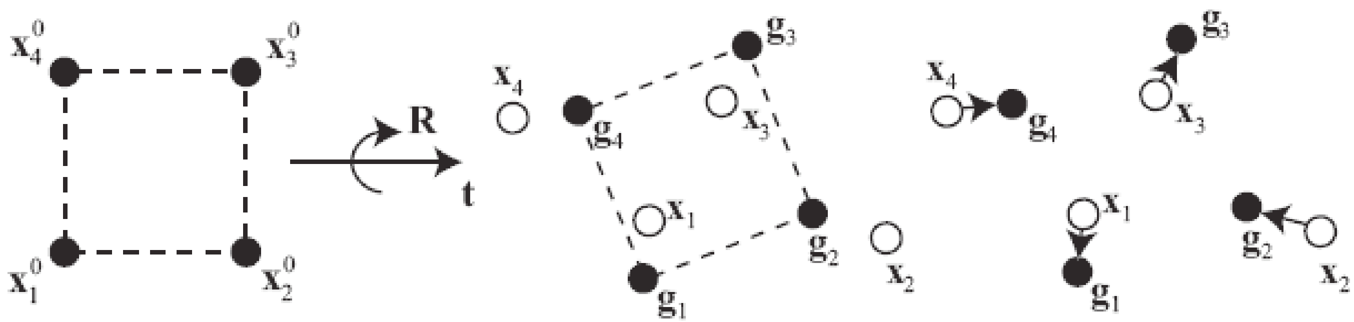
- Non-physically based
  - Ignore what really happens in the physical world
  - Come up with a function for computing internal forces
    - Based on current position and velocity
  - Examples
    - Mass-Spring Models [4]
    - A Versatile and Robust Model for Geometrically Complex Deformable Solids [5]
    - Meshless Shape Matching [6] *

Pros and Cons

- Physically Based
  - Pros:
    - More correct
      - Can be used for prediction
    - Parameters from real objects
  - Cons:
    - Messy math
    - Hard to implement
    - More expensive
- Non-Physically Based
  - Pros:
    - Easier to implement
    - Cheaper
    - Easier math
  - Cons:
    - Lots of parameters
    - Parameters make less sense
    - Can't get parameters from real objects
    - Can't use to predict

Meshless Shape Matching Basics

- Deformable Objects consist of lots of particles
- Match current object shape against the rest shape
  - Start with best fit rigid transformation
- Pull particles toward the matched shape
  - Can update a particle velocity and position independently
    - Need not care about other particles



Best fit Rigid Transformation

- Find **R** and **t** to minimize error:

Current position relative to **t**

$$\sum_i m_i \left\| \mathbf{R}(\mathbf{x}_i^0 - \mathbf{x}_{cm}^0) - (\mathbf{x}_i - \mathbf{t}) \right\|^2$$

Position relative to CM of the best fit rigid transformation

  - $\mathbf{x}_i^0$ – Rest position of particle i
  - $\mathbf{x}_i$ – Current position of particle i
  - $\mathbf{x}^0_{cm}$ – Center of mass of particles at rest configuration
  - m_i – Mass of particle i

- Best fit **t** is just the center of mass of current particles' position
  - Match with intuition

$$\mathbf{t} = \mathbf{x}_{\mathrm{cm}} = \frac{\sum_i m_i \mathbf{x}_i}{\sum_i m_i}$$

Best fit Rigid Transformation

- Computing **R** (Optimum Rotation)
  - First, remove translation from consideration
    - Rewrite the optimization equation

$$\sum_i \mathbf{m}_i (\mathbf{A}\mathbf{q}_i - \mathbf{p}_i)^2$$

    - Where,
      - **A** is a 3x3 matrix, a linear transformation
      - $\mathbf{q}_i = \mathbf{x}_i^0 - \mathbf{x}_{cm}^0$, rest position relative to the rest center of mass
      - $\mathbf{p}_i = \mathbf{x}_i - \mathbf{x}_{cm}$, current position relative to the current center of mass
  - Compute best fit **A**
    - Turn out to be $\mathbf{A} = (\sum_i m_i \mathbf{p}_i \mathbf{q}_i^T)(\sum_i m_i \mathbf{q}_i \mathbf{q}_i^T)^{-1} = \mathbf{A}_{pq}\mathbf{A}_{qq}$
  - Extract Rotation Part
    - Linear Transformation = Rotation + Scaling + Shear
    - **A** = **RS,** **R** is a rotation mat, **S** is a symmetric mat

nVIDIA.

Extracting Rotation

- We know that **A** = **RS**
- Can show that **S** = sqrt$(\mathbf{A}^T\mathbf{A})$, eg. $\mathbf{A}^T\mathbf{A} = \mathbf{SS}$
- We can then get $\mathbf{R} = \mathbf{AS}^{-1}$
- Computing $\mathbf{S}^{-1}$
  - Find $\mathbf{Q} = \mathbf{A}^T\mathbf{A}$
  - Diagonalize **Q**, $\mathbf{Q} = \mathbf{J}^T\mathbf{DJ}$
    - With Jacobi Rotation
  - Compute $\mathbf{S}^{-1} = \mathbf{J}^T\text{sqrt}(\mathbf{D}^{-1})\mathbf{J}$
    - sqrt$(\mathbf{D}^{-1})$ is just matrix of 1/sqrt of diagonal entries of **D**
- Paper suggests extracting R from A_{pq}
  - Bad idea because A_{pq} is ill-conditioned
  - Plus we're working with single precision float here

Jacobi Rotation

```
void Jacobi(inout float3x3 mat, inout float3x3 jmat, in int j, in int k) {
	// First, check if entries (j,k) is too small or not, if so, do nothing
	if (abs(mat[j][k]) > 1e-20) {
		// This is just some math to figure out cosine and sine necessary to zero out the two entries
		float tau = (mat[j][j]-mat[k][k])/(2.0f*mat[j][k]);
		float t = sign(tau) / (abs(tau) + sqrt(1 + tau*tau));
		float c = 1/sqrt(1+t*t);
		float s = c*t;
		// Build the rotation matrix
		float3x3 R = {1.0f, 0.0f, 0.0f,  0.0f, 1.0f, 0.0f,  0.0f, 0.0f, 1.0f};
		R[j][j] = c; R[k][k] = c; R[j][k] = -s;  R[k][j] = s;

		jmat = mul(jmat, R); mat = mat*R;
		R[j][k] = s; R[k][j] = -s;
		mat = R*mat;
	}
}

float3x3 ComputeOptimumRotation(in float3x3 A) {
	float3x3 jmat = {1.0f, 0.0f, 0.0f,  0.0f, 1.0f, 0.0f,   0.0f, 0.0f, 1.0f};
	float3x3 mat = mul(transpose(A), A);

	// Do 5 iterations of Jacobi rotation
	[unroll(5)] for (int i = 0; i < 5; i++) {Jacobi(mat, jmat, 0, 1); Jacobi(mat, jmat, 0, 2);Jacobi(mat, jmat, 1, 2);}
	// A^tA == jmat^t mat jmat
	// OptimumR = A jmat^t sqrt(1/mat) jmat
	float3x3 optimumR = transpose(mul(A, mul( transpose(jmat), float3x3(
				jmat[0] / sqrt(mat[0][0]), jmat[1] / sqrt(mat[1][1]), jmat[2] /sqrt(mat[2][2])))));
	const int first = 1, second = 2, third = 0;
	optimumR[first] = normalize(optimumR[first]);
	optimumR[third] = normalize(cross(optimumR[first], optimumR[second]));
	optimumR[second] = cross(optimumR[third], optimumR[first]);
	return transpose(optimumR);
}
```

Particles position and velocities update

- Compute intermediate position and velocity

$$\bar{\mathbf{v}}_i = \mathbf{v}_i + h\mathbf{f}_i / m_i$$

$$\bar{\mathbf{x}}_i = \mathbf{x}_i + h\bar{\mathbf{v}}_i$$

  - f_i is the force acting on particle i
    - Eg. Gravity, Collision Force, User Specified Force
- Compute best fit rigid transformation of the intermediate position
- Update the position and velocity

$$\mathbf{g}_i = \mathbf{R}\mathbf{q}_i + \bar{\mathbf{x}}_{cm} \qquad \mathbf{q}_i = \mathbf{x}_i^0 - \mathbf{x}_{cm}^0$$

$$\mathbf{v}'_i = \mathbf{v}_i + h\mathbf{f}_i / m_i + \frac{\alpha}{h}(\mathbf{g}_i - \bar{\mathbf{x}}_i)$$

$$\mathbf{x}'_i = \mathbf{x}_i + h\mathbf{v}'_i$$

- α control how fast the deformable body restore to rigid shape
  - α = 1 will make this a rigid body simulation

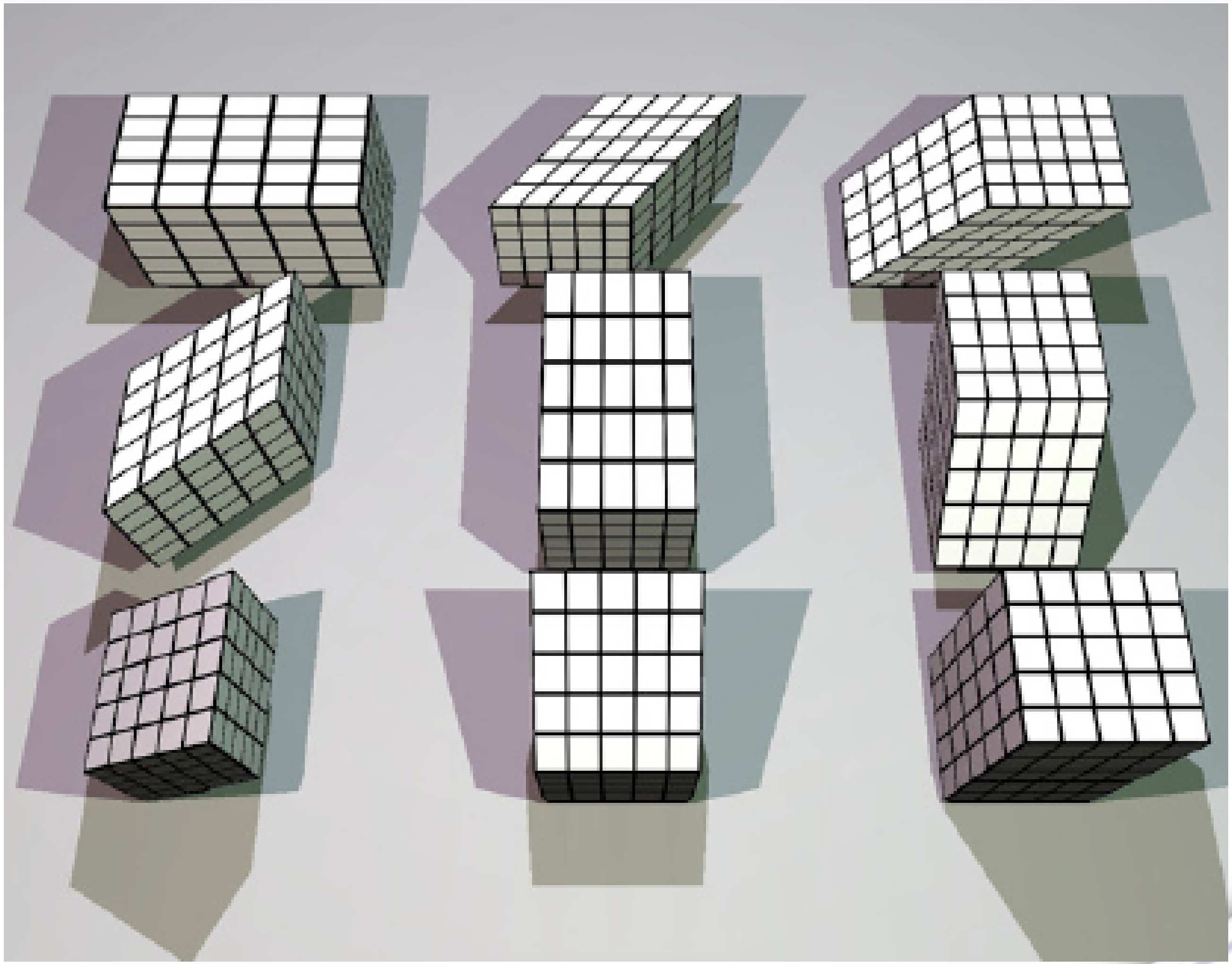
Extension

- So far, goal shape is always a rigid transformation
  - Will support only small deformations
- To obtain a more interesting deformation:
  - Want to make the goal shape be a deformed configuration
  - Then slowly pull the goal shape towards the rigid transformation
- Blend rigid transformation with linear transformation
  - **A** is the best fit
    - To conserve volume, divide **A** by $\sqrt[3]{det(\mathbf{A})}$
  - Use β **A + (1-** β**) R** in place of **R** in computing the goal position

$$\mathbf{g}_i = (\beta\mathbf{A} + (1-\beta)\mathbf{R})\mathbf{q}_i + \bar{\mathbf{x}}_{cm}$$

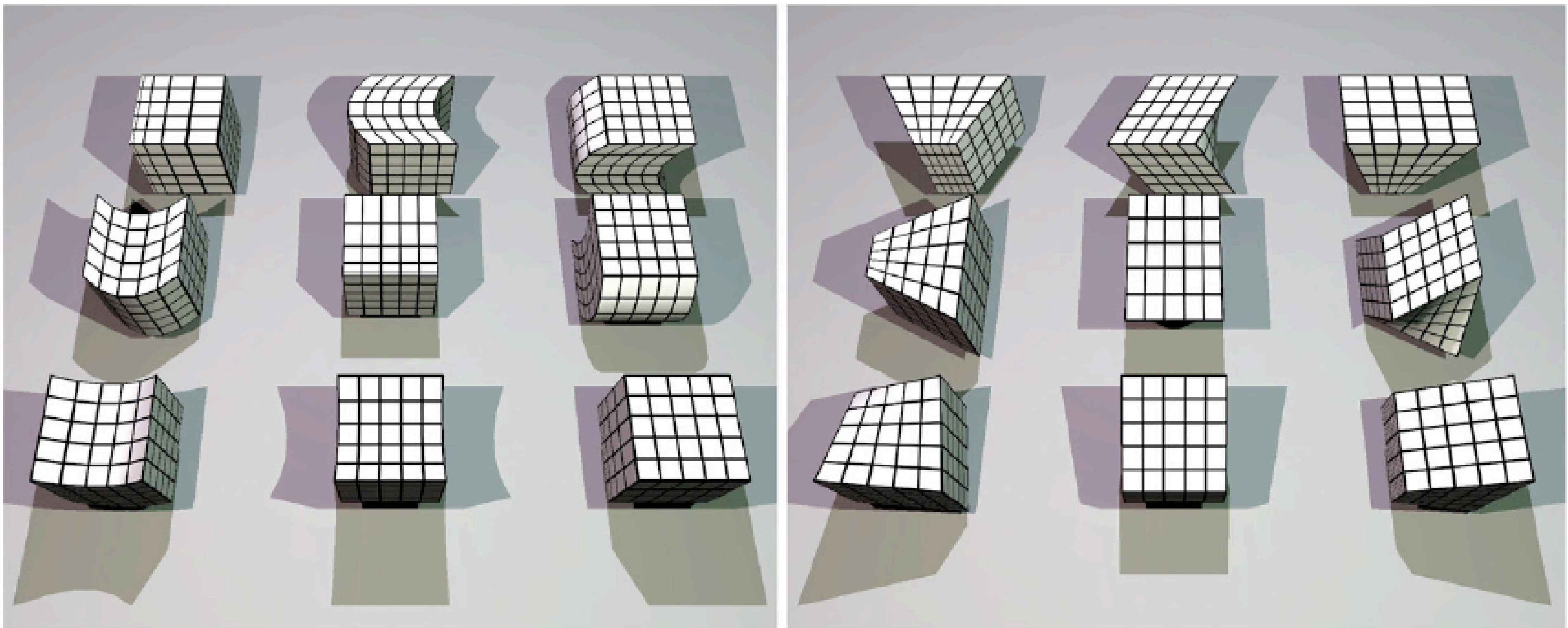
  - **β** must be < 1 so as to have tendency to restore to rest state

Extension



More Extension

- Linear not good enough
  - Use quadratic best fit!



Best Fit Quadratic Transformation

- Best fit quadratic transformation

$$\overline{\mathbf{A}} = [\mathbf{AQM}] \in \Re^{3\times 9}$$

  - **A** is linear transformation
  - **Q** is pure quadratic terms
  - **M** is mixed quadratic terms

- When $\sum_i m_i(\overline{\mathbf{A}}\overline{\mathbf{q}}_\mathbf{i} - \mathbf{p_i})^2$ is minimized where

$$\overline{\mathbf{q}} = [q_x, q_y, q_z, q_x^2, q_y^2, q_z^2, q_x q_y, q_y q_z, q_z q_x]^T \in \Re^{1x9}$$

- The minimum turns out to be:

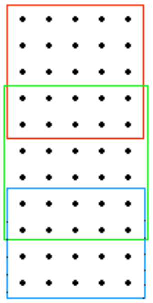
$$\overline{\mathbf{A}} = (\sum_i m_i \mathbf{p_i} \overline{\mathbf{q}}_i^T)(\sum_i m_i \overline{\mathbf{q}}_\mathbf{i} \overline{\mathbf{q}}_\mathbf{i}^T)^{-1} = \overline{\mathbf{A}}_\mathbf{pq} \overline{\mathbf{A}}_\mathbf{qq}$$

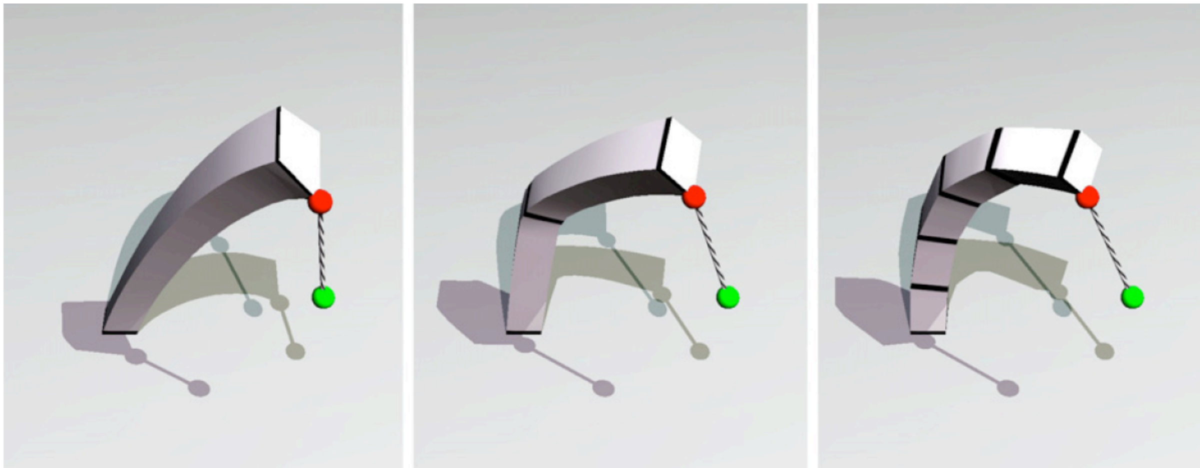
- Then use $\mathbf{g}_i = (\beta\overline{\mathbf{A}} + (1-\beta)\overline{\mathbf{R}})\overline{\mathbf{q}}_i + \overline{\mathbf{x}}_{cm}$ to compute goal shape
- $\overline{\mathbf{A}}_{qq} \in \Re^{9x9}$ Can be pre-computed $\overline{\mathbf{R}} \in \Re^{3\times 9} = [\mathbf{R00}]$

NVIDIA.

Cluster Based Deformation

- Deformation for large complex objects may not be well fitted by a single quadratic deformation
- Cluster particles together
  - Particles can be in several clusters
  - Each cluster computes a separate goal shape
- Goal shapes from clusters are then averaged to form final goal shape



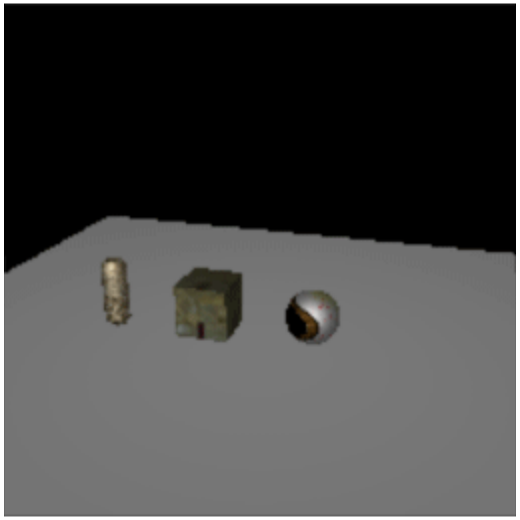


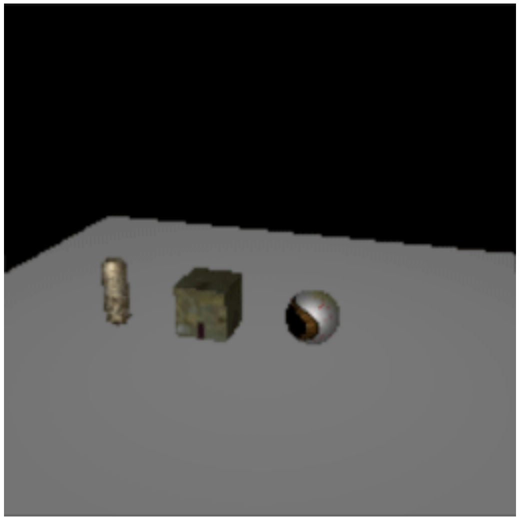
GeForce 8800 Implementation

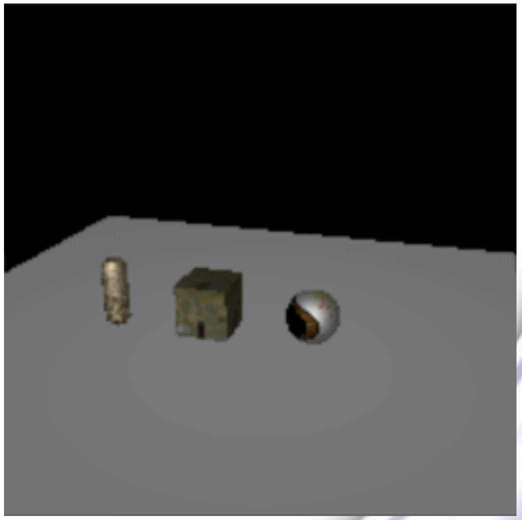
- Goals:
  - Fast deformation physics for objects with multiple clusters
  - Perform collision detection and handling
  - Done entirely on GPU
  - Lots of objects in real time
  - Support skinning
    - Simulate low-resolution mesh
    - Render high resolution mesh

Demo

- Falling Objects
  - Varying α, β







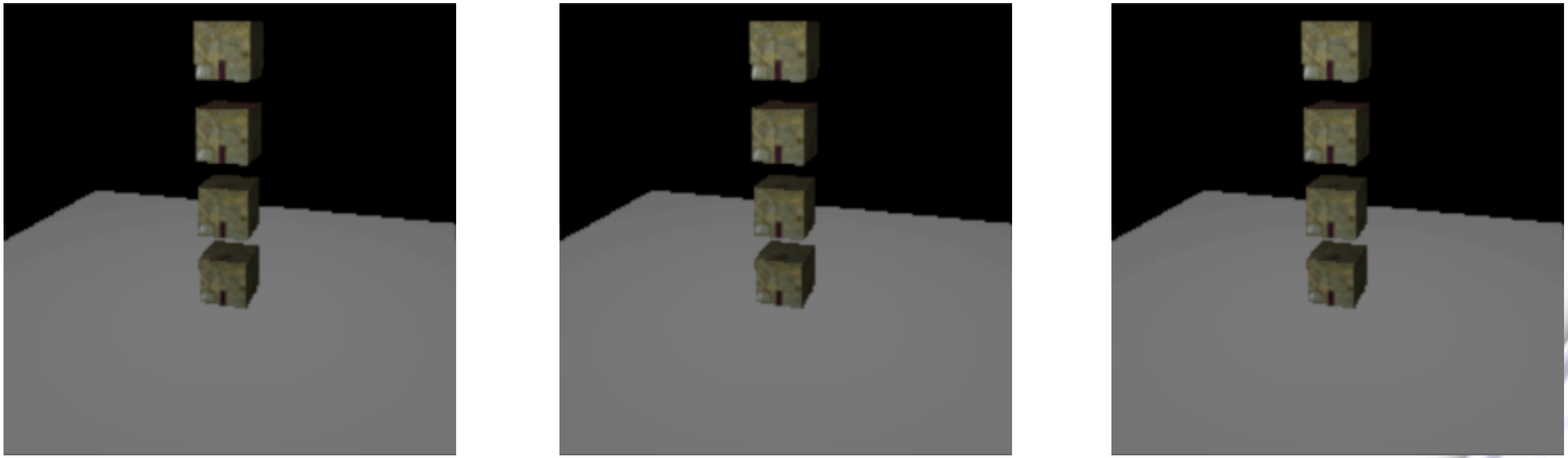
Demo

- Collision with height map
  - Varying α, β



Demo

- Collision between objects
  - Varying α, β



Considerations

- Need to perform computations in parallel manner
  - Doing one pass for all objects before doing the next pass

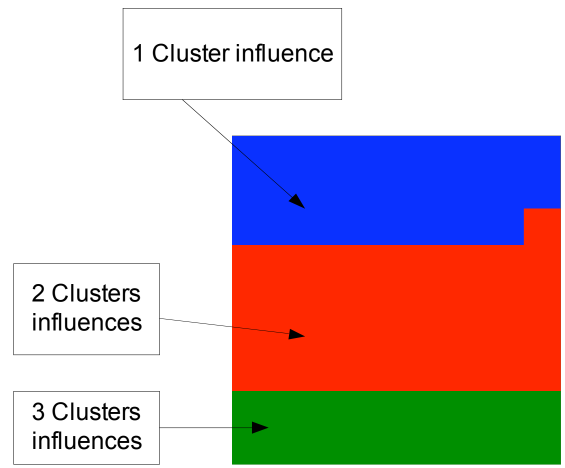
- Balance between having small number of passes and having redundant computations

Data Structure

- 3 types of Texture2Ds used
  - For storing information about each particle
  - For storing information about particles in each cluster
    - A particle can belong to many clusters
  - For storing information about clusters
- 2 types of usage
  - Never changes during run-time
  - Being updated and used dynamically

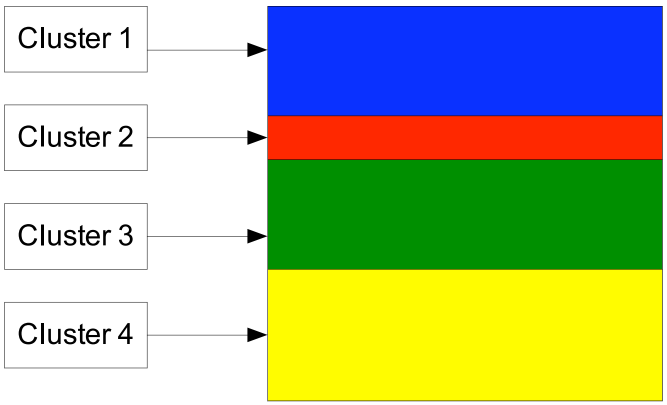
Data Structure

- Texture2Ds for storing information about particles,
  - Current Position and Intermediate Position, xTex, xBarTex
    - XYZ→ RGB, Mass→A
  - Current Velocity, vTex
    - XYZ→ RGB, #influenced cluster→A
  - Acceleration, aTex
    - XYZ→ RGB
  - Goal Position, gTex
    - XYZ→RGB
  - $\bar{q}$, qBarTex
    - 3 Texels $\bar{\mathbf{q}} = [\mathbf{q}_x, \mathbf{q}_y, \mathbf{q}_z, \mathbf{q}_x^2, \mathbf{q}_y^2, \mathbf{q}_z^2, \mathbf{q}_x\mathbf{q}_y, \mathbf{q}_y\mathbf{q}_z, \mathbf{q}_z\mathbf{q}_x]^T$
  - Particles are sorted
    - Row major order
    - Based on number of clusters that influence them



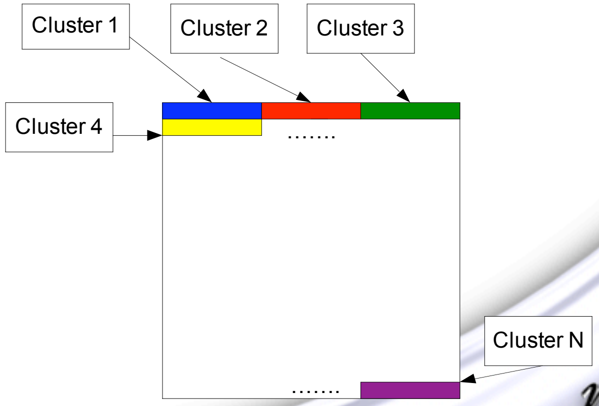
Data Structure

- Texture2Ds for storing information about particles in each cluster
  - Pointer to xTex texture, xAdrTex
    - To specify which particles are members of this cluster
  - Position of particles, xValTex
    - To reduce # of dependent texture fetch
  - Position of particles wrp to cluster CM, pValTex
- Each cluster corresponds to a quad in the texture



Data Structure

- Texture2Ds for storing information about clusters
- Take up various number of texels
  - CM, cmTex, takes 1 texel per cluster
    - X,Y,Z→RGB, Total Mass→ A
  - ApqbarTex, takes 8 texels
    - Packed 3x9 matrix
  - Goal Transformation, transformTex, takes 8 texels
    - Packed 3x9 matrix
  - AqqbarTex, take 12 texels
    - Packed symmetric 9x9 matrix



Texture Summary

- Particle info
  - xTex – Current particle position
  - xBarTex – Intermediate particle position
  - vTex – Current particle's velocity
  - aTex – Current particle's acceleration
  - gTex – Particle's goal position
  - qBarTex – Particle $\bar{q}$
- Particle in cluster info
  - xAdrTex – Pointer to fetch particle position
  - xValTex –Cluster particle's current position
  - gValTex – Cluster particle's goal position
  - pValTex – Cluster particle's position wrt to CM
  - aValTex – Cluster particle's acceleration
- Cluster info
  - cmTex – Cluster's center of mass
  - ApqbarTex - Cluster's ApqBar
  - transformTex – Transformation for computing goal
  - AqqbarTex - Cluster's AqqBar

NVIDIA.

Example

- **6 Particles**
- **2 clusters**
  - **Cluster 0 has particles 0 1 2 3**
  - **Cluster 1 has particles 2 3 4 5**

xTex

x0	x1	x4
x5	x2	x3

vTex

v0	v1	v4
v5	v2	v3

qBarTex

qBar0	qBar1	qBar4
qBar5	qBar2	qBar3

xAdrTex

00	01	11	12
11	12	02	10

xValTex

x0	x1	x2	x3
x2	x3	x4	x5

cmTex

c0	c1

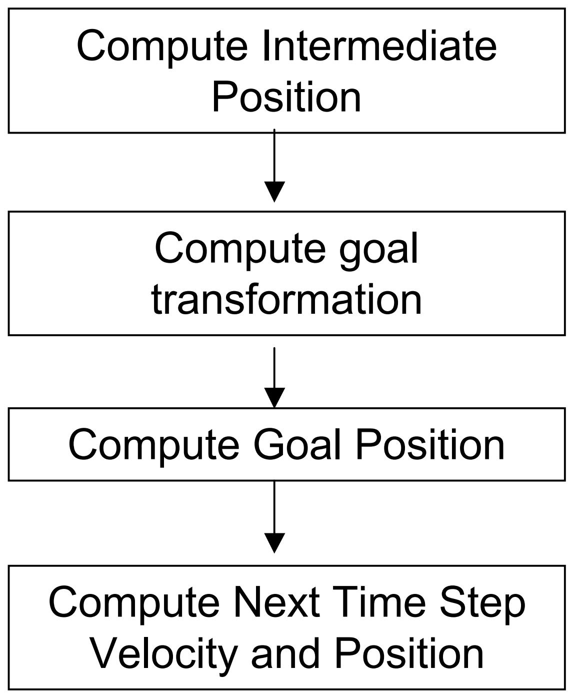
transformTex

Transform0	Transform1

Overview of DX10 implementation

No collision

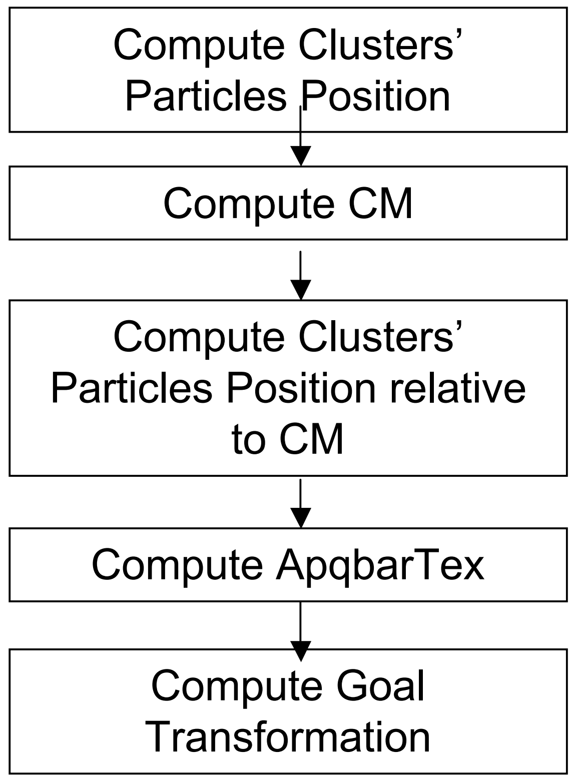
No skinning



Computing Intermediate Position

- Input: xTex, vTex, aTex, Height Map
- Output: xBarTex $\bar{\mathrm{x}}(t) = \mathrm{x}(t) + h\bar{\mathrm{v}}(t),$ $\bar{\mathrm{v}}(t) = \mathrm{v}(t) + h\mathbf{f}_{ext}(t)/m_i$
- Computation: PS
  - Draw a quad
  - First compute intermediate velocity
  - Then compute intermediate position
  - Acceleration includes:
    - Gravity
    - External force
    - Collision force with height map
      - Fetch height from height map (RGB encodes normal, A encodes height)
      - See if it penetrates ground or not
      - If so, apply force in heightmap's normal direction
    - Collision force with other objects (later)

Computing Goal Transformation

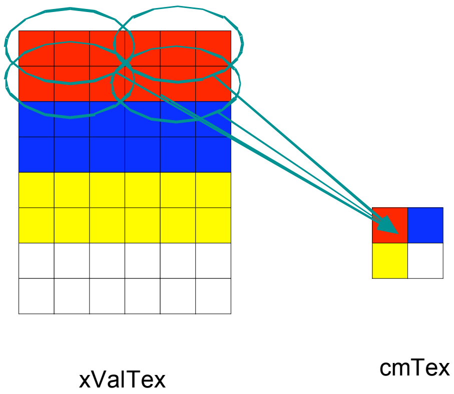


Computing Clusters' Particles Position

- Compute position of particles for each cluster
- Input: xBarTex, xAdrTex
- Output: xValTex
- Computation: PS
  - Draw quads, one per cluster
  - Fetch xAdrTex to get pointer to xBarTex
  - Fetch xBarTex and output

Computing CM

- Compute center of mass for each cluster
- Input: xValTex
- Output: cmTex
- Computation: VS, PS
  - Draw points, several points per cluster
  - Each point sum the position of M particles weighted by the mass, fetched from xValTex
  - For points belonging to the same cluster, output to the same pixel
  - Use 32-bit float additive alpha blending
    - GeForce 8800 has this functionality!



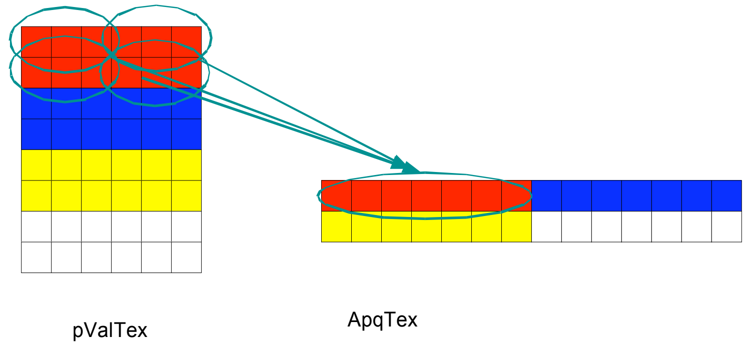
Computing positions relative to CM

- Input: xValTex, cmTex
- Output: pValTex
- Computation: GS, PS
  - Draw points, one point per cluster
  - GS:
    - Fetches cmTex of the cluster
    - Create a quad to cover portion of pValTex that corresponds to the cluster
  - PS fetches xValTex and subtract with CM

Computing ApqBarTex

$$\overline{\mathbf{A}}_{pq} = \sum_i \mathbf{m}_i \mathbf{p}_i \overline{\mathbf{q}}_i^T$$

$$\overline{\mathbf{A}}_{pq} = \begin{bmatrix} x_{1r} & x_{1g} & x_{1b} & x_{1a} & x_{2r} & x_{2g} & x_{2b} & x_{2a} & x_{7r} \\ x_{3r} & x_{3g} & x_{3b} & x_{3a} & x_{4r} & x_{4g} & x_{4b} & x_{4a} & x_{7g} \\ x_{5r} & x_{5g} & x_{5b} & x_{5a} & x_{6r} & x_{6g} & x_{6b} & x_{6a} & x_{7b} \end{bmatrix}$$

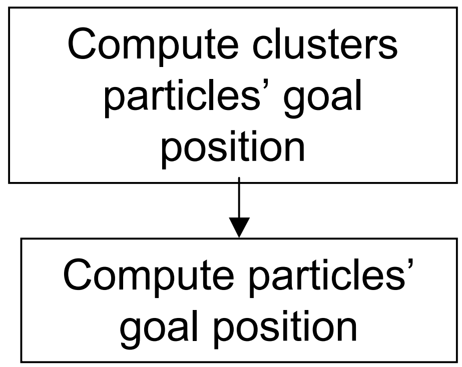


- Input: pValTex, qBarTex
- Output: ApqBarTex
- Computation: GS (can push up to VS)
  - Draw points, several points per cluster
  - Compute $\mathbf{m}_i \mathbf{p}_i \overline{\mathbf{q}}_i^T$, which is a 3x9 matrix in GS
    - Sum contribution from M particles
  - Output 7 adjacent points
  - Use 32 bits float additive alpha blending to sum the sums

Computing Goal Transformation

- Input: ApqBarTex, AqqBarTex
- Output: transformTex
- Computation: GS (can push up to VS)
  - Draw points, 1 point per cluster
  - Compute $\overline{\mathbf{A}}$ by multiplying $\overline{\mathbf{A}}_{pq}$ with $\overline{\mathbf{A}}_{qq}$
    - Expand the packed $\overline{\mathbf{A}}_{qq}$
  - Extract the 3x3 left sub matrix to get **A**
  - Compute optimum rotation, **R,** with Jacobi Method
  - Compute $\mathbf{T} = \beta\overline{\mathbf{A}} + (1-\beta)\overline{\mathbf{R}}$
  - Output 7 points

Computing Goal Position



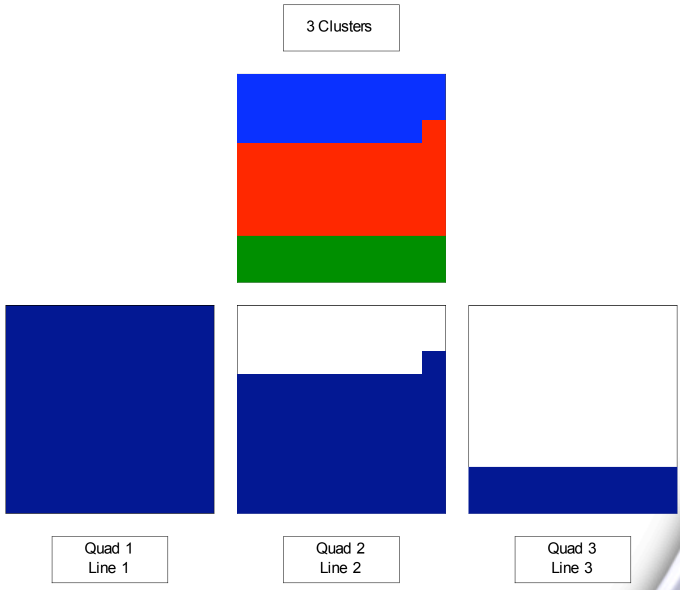
Computing Clusters Particles' Goal Position

- Compute the goal position of particles in each cluster
- Input: transformTex, pValTex, cmTex, qBarTex
- Output: gValTex
- Computation: GS, PS
  - Render quads, 1 quad per cluster
  - Use GS to fetch cmTex, transformTex and generates quad
  - Use PS to fetch qBarTex, multiply with the transformation and add with CM

$$\mathbf{g}_i = \mathbf{T}\overline{\mathbf{q}}_i + \overline{\mathbf{x}}_{cm}$$

Computing Particles' Goal Position

- Compute goal positions of particles
  - Average the goal position of the particle from the cluster it belongs to
- Input: gValTex
- Output: gTex
- Computation: PS
  - Draw quads and lines
    - First quad and a line for all particles with >=1 influence cluster
    - Next quad and 2 lines for all particles with >=2 influence clusters
    - …..
  - Do additive alpha blending
- This is why we sort the particles based on the number of influences



Compute Next Time Step Position & Velocity

- Update the position and velocity of particles
- Input: xTex, vTex, aTex, gTex, xBarTex
- Output: xTex', vTex'
- Computation: PS
  - Draw a quad
  - Use MRT, for position and velocity
  - Compute velocity first then use it to compute position

$$v_i(t+h) = v_i(t) + \alpha \frac{g_i(t) - \bar{x}_i(t)}{h} + hf_{ext}(t)/m_i$$

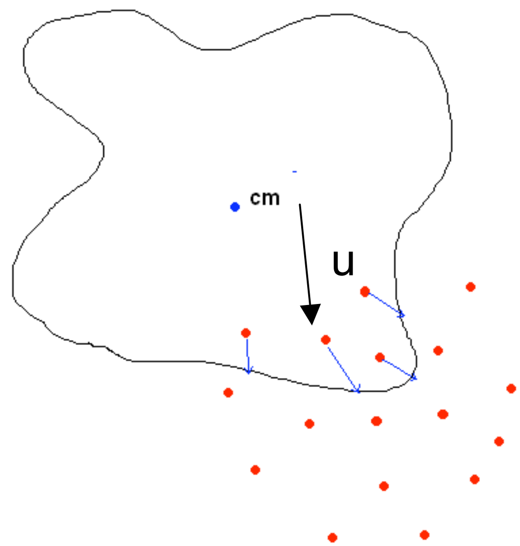
$$x_i(t+h) = x_i(t) + hv_i(t+h)$$

Collision Handling

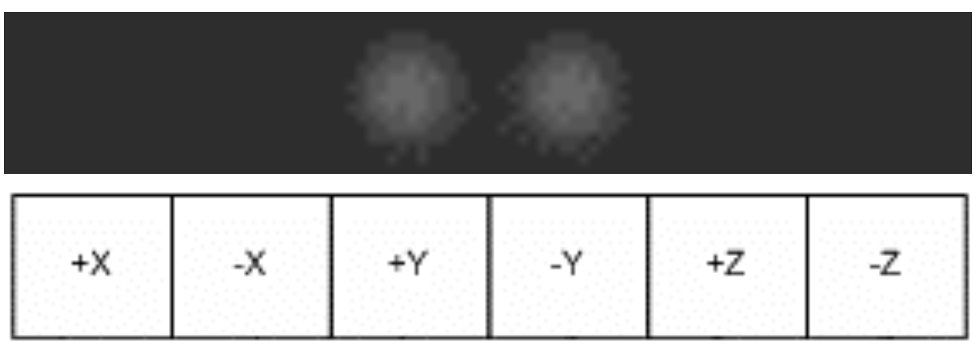
- Collision detection with depth cube map
  - Detect if particles in a cluster penetrate through another cluster or not
  - If so, apply penalty force
- For a cluster,
  - Need to check if particles collide with any other cluster or not
  - Slow, $O(N^2)$ cube map look up
  - Need some pruning
    - Only check clusters whose bounding box overlaps with this cluster

Collision Detection with Depth Cube Map

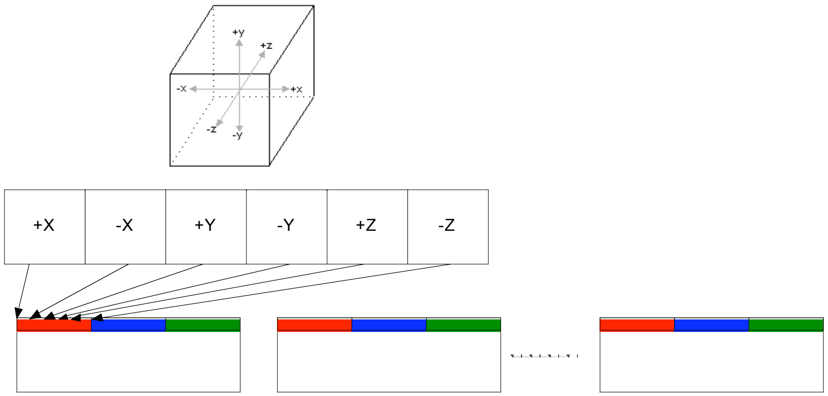
- Create depth cube map for each cluster
  - Centered at CM
  - Update every frame
  - Low Resolution, use 16x16 now



- Look into depth cube map in direction **u**
  - If distance from CM < depth
    - Apply force in direction of **u**
    - Magnitude proportional to depth-distance from CM

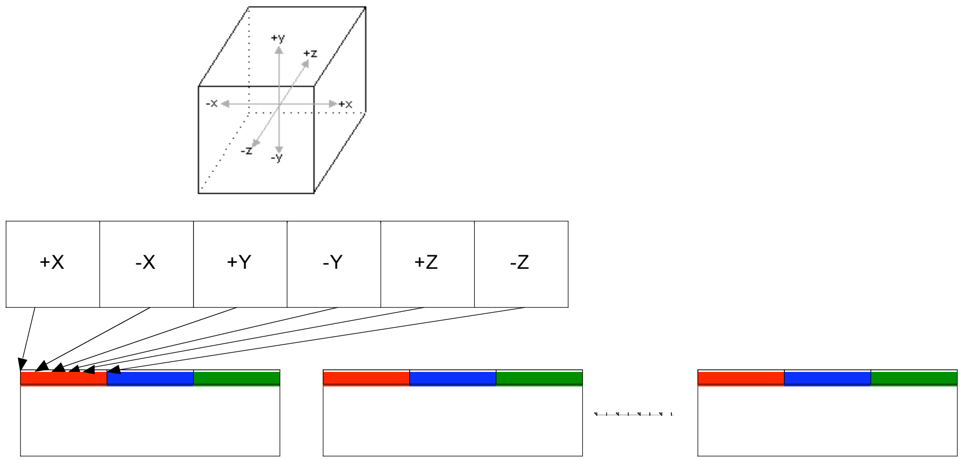


Cube Map Collision Detection Implementation



- DX10 does not support array of cube maps
  - Instead flatten the cube map and stores the 6 faces in a Texture2D slice
  - Store several cube maps per Texture2D slice
- Use a cube map atlas
  - Store a 2D texture coordinate in the cube map
  - Look up the cube map atlas to get (u,v)
  - Offset u,v and choose slice # appropriately to fetch the correct cube map
  - Fetch the corresponding position in the Texture2D slice

Cube Map Creation

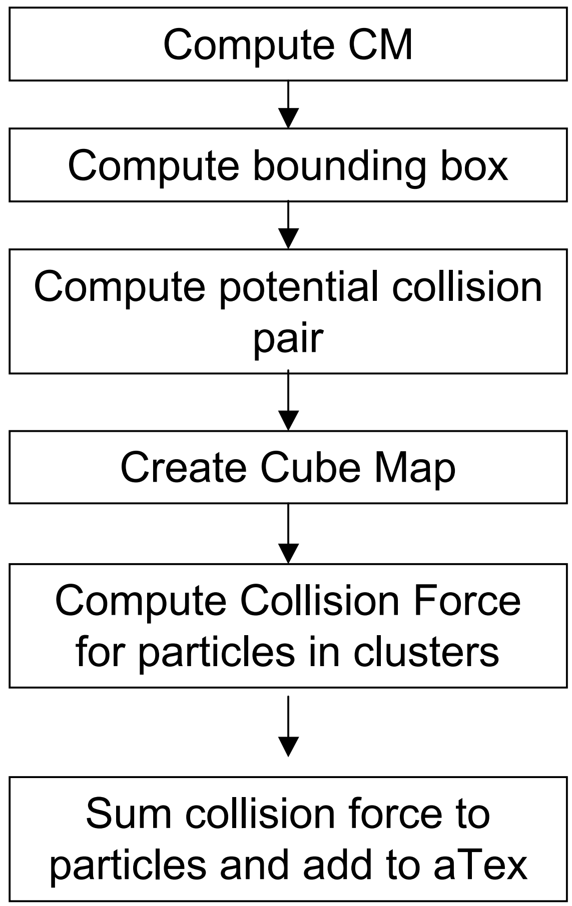


- DX10 allows only limited numbers of textures that can be used at a time
- Suppose there are N clusters and the Texture2DArray is of size S,
  - Need N/S rendering passes
  - Each pass create S cube map
    - Use GS to output 6 triangles per each input triangle
      - Output to 6 viewports of the same Texture2D slice
      - Choose Texture2D slice depending on which cluster the triangle belongs to
    - Change viewport after every pass

Pruning

- Don't want to do $O(N^2)$ cube map lookup
  - Compute Bounding Box of clusters
  - Do cube map check only for pairs of clusters whose BBs overlap
  - Avoid checking pairs of clusters from the same object
  - For each pair (i, j)
    - For all particles in cluster i, lookup into the depth cube map of cluster j
    - Apply penalty force to particle i if found to penetrate

Collision Handling Overview



Same as before

Similar to CM, but use Max, Min

Similar to averaging the goal position

Computing Potential Colliding Pairs

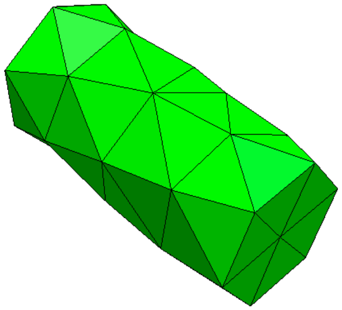
- Input: Bounding Boxes(Maxs and Mins of xyz of particles in each cluster)
- Output: Potential Colliding Pairs
- Computation: GS stream out (can push to VS)
  - Bind NULL vertex buffer
  - Draw all possible (i, j) where cluster i and j do not come from the same object and i < j
  - If bounding box of i, j overlap
    - Stream out 2 points containing information about (i, j) and (j, i)
  - Can later use more sophisticated pruning techniques
  - We store the ID of the object each cluster belong to in a constant buffer

Computing Collision Force

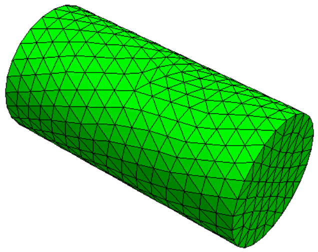
- Input: Potential Colliding Pairs, cmTex, pValTex
- Output: aValTex
- Computation: GS, PS
  - Use DrawAuto to draw points of potentially colliding pairs (i,j)
  - In GS,
    - Turn a point to a quad covering particles in cluster i
    - Fetch CM of cluster j and pass as a vertex attribute
  - In PS, computation is done for each particles in i
    - Look up cube map of j and check for penetration
    - Apply force proportional to penetration depth
      - In direction radially outward from CM of j
  - Additive alpha blending to sum force

Skinning

- Treat particles as control points
  - Compute surface mesh's vertices based on control point position
- Barycentric interpolation for now
  - Weights stored in a texture
  - 4 control points per vertex
- Need tetrahedral mesh that encloses and approximates the surface mesh
  - Generate with NetGen
- Given a tetrahedral mesh and a surface mesh:
  - Program will figure out which tetrahedron each of the vertices of the surface mesh are in



Coarse Tetrahedral Mesh for Simulation



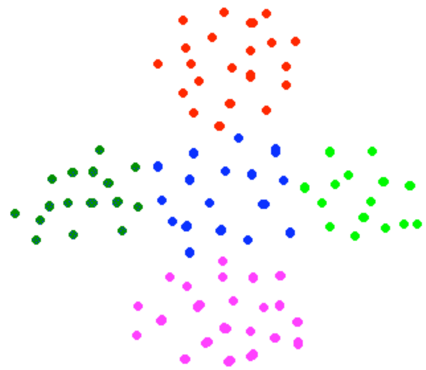
Detailed Surface Mesh for Rendering

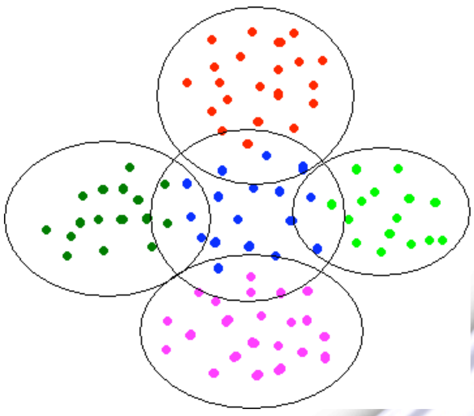
Normal vector computation

- Use GS and Alpha blending
- Input: Deformed vertex positions as a texture
- Output: Normal vectors as a texture
- Computation:
  - GS:
    - Compute triangle's area weighted normal
    - Turn a triangle into 3 points each with normal as color
    - Output 3 points to the corresponding vertices position. Use additive alpha blending to accumulate vertex normal
- Normalize it before use
- Use vertex texture fetch to get the normal out

Automatic Cluster Generation

- Given the tetrahedral mesh,
  - Compute K-Mean of the vertices
  - Partition the vertices into K groups
  - Make each group a cluster
  - Also add 1-Ring neighbors to clusters
- Done in preprocessing step on the CPU





Current Status

- Currently 20 Computation Passes + 1 Rendering Pass
- Load X files and .mesh file (from NETGEN)
- Parameters for each objects:
  - α, β for controlling softness
  - Penalty force constant
    - In collision event between (i, j), will take the max
  - Number of clusters to use

Result

Future

- Plastic deformation (permanent deformation)
  - Need to update $\overline{\mathbf{A}}_{\mathbf{qq}}$ on the fly
  - Need 9x9 symmetric matrix inversion in GPU
    - Gaussian Elimination in GS?
- Other solid simulation models
  - FEM
    - Need sparse linear system solver
- Smarter collision pruning
- More sophisticated collision handling
  - Contact surface approximation with cube map?

References

- 1. Interactive Deformations Using Modal Analysis by Hauser, K., Shen, C., O'Brien, J. F.
- 2. Interactive virtual materials, Matthias Muller, Markus Gross
- 3. Real-Time Subspace Integration of St.Venant-Kirchhoff Deformable Models, Jernej Barbic and Doug L. James
- 4. Google “mass spring model”
- 5. A Versatile and Robust Model for Geometrically Complex Deformable Solids, M. Teschner, B. Heidelberger, M. Mueller, M. Gross
- 6. Meshless Deformations Based on Shape Matching M. Mueller, B. Heidelberger, M. Teschner, M. Gross