Extensions to Progressive Meshes

Ranjitha Kumar
Papers

- View-Dependent Refinement of Progressive Meshes
  Hugues Hoppe
  SIGGRAPH 1997

- Progressive Simplicial Complexes
  Jovan Popovic, Hugues Hoppe
  SIGGRAPH 1997
View-Dependent Refinement

✦ Motivation: regions of the mesh closer to the viewer should be more refined than the regions farther away.

✦ Contribution: real-time framework for selectively refining an arbitrary PM according to changing view parameters, using efficient refinement criteria
PM Review

✦ Edge collapse sequence:

\[(\hat{M} = M^n) \xrightarrow{ecol_{n-1}} \ldots \xrightarrow{ecol_1} M^1 \xrightarrow{ecol_0} M^0 .\]

✦ Vertex split sequence:

\[M^0 \xrightarrow{vsplit_0} M^1 \xrightarrow{vsplit_1} \ldots \xrightarrow{vsplit_{n-1}} (M^n = \hat{M}) .\]

✦ PM representation:

\[(M^0, \{vsplit_0, \ldots, vsplit_{n-1}\})\]
Modified Framework

- Built upon an arbitrary PM with modified definitions of vsplit (vertex split) and ecol (edge collapse)
Vertex Hierarchy

- A selectively refined mesh corresponds to a “vertex front” through the hierarchy
Preconditions

A vertex or face is active if it exists in the selectively refined mesh.

**Preconditions** We define a set of preconditions for vsplit and ecol to be legal (refer to Figure 4).

A vsplit($v_s, v_t, v_u, \ldots$) transformation is legal if

1. $v_s$ is an active vertex, and
2. the faces $\{f_{n0}, f_{n1}, f_{n2}, f_{n3}\}$ are all active faces.

An ecol($v_s, v_t, v_u, \ldots$) transformation is legal if

1. $v_t$ and $v_u$ are both active vertices, and
2. the faces adjacent to $f_t$ and $f_r$ are $\{f_{n0}, f_{n1}, f_{n2}, f_{n3}\}$, in the configuration of Figure 2.
Implementation

```
struct ListNode {
    ListNode* next; // Node possibly on a linked list
    ListNode* prev; // 0 if this node is not on the list
};

struct Vertex {
    ListNode active; // list stringing active vertices V
    Point point;
    Vector normal;
    Vertex* parent; // 0 if this vertex is in M^0
    Vertex* vt; // 0 if this vertex is in M; (v_u=v_t+1)
    // Remaining fields encode vsplit information, defined if vt \neq 0.
    Face* fr; // (fr=fl+1)
    Face* fn[4]; // required neighbors f_{n0},f_{n1},f_{n2},f_{n3}
    RefineInfo refine_info; // defined in Section 4
};

struct Face {
    ListNode active; // list stringing active faces F
    int matid; // material identifier
    // Remaining fields are used if the face is active.
    Vertex* vertices[3]; // ordered counter-clockwise
    Face* neighbors[3]; // neighbors[i] across from vertices[i]
};

struct SRMesh {
    // Selectively refinable mesh
    Array<Vertex> vertices; // set V of all vertices
    Array<Face> faces; // set F of all faces
    ListNode active_vertices; // head of list V \subseteq V
    ListNode active_faces; // head of list F \subseteq F
};
```

Figure 5: Principal C++ data structures.
function qrefine(v_s)

   // Refine only if it affects the surface within the view frustum.
   if outside_view_frustum(v_s) return false

   // Refine only if part of the affected surface faces the viewer.
   if oriented_away(v_s) return false

   // Refine only if screen-projected error exceeds tolerance $\tau$.
   if screen_space_error(v_s) $\leq \tau$ return false

   return true
View Frustum

- Basic Idea: the surface outside of the view frustum should be coarse.
- Compute bounding spheres for each vertex that encompasses all of its descendant vertices.
- Check if the bounding sphere lies completely outside the view frustum
- For a sphere of radius $r_v$, centered at $\mathbf{v} = (v_x, v_y, v_z)$ lies outside the frustum if

$$a_i v_x + b_i v_y + c_i v_z + d_i < -r_v \quad \text{for any } i = 1 \ldots 4$$

where each linear functional $a_i x + b_i y + c_i z + d_i$ measures the signed Euclidean distance to a side of the frustum.
Hierarchical Method

- Step 1
- Step 2
- Step 3
Result
Surface Orientation

- Hierarchically compute for each vertex a cone about its normal that bounds the normal space for its descendants.
Results
Screen-space Geometric Error

- Screen projected distance between approximate mesh and original mesh is everywhere less than a scree-space tolerance $\tau$.
- Need a measure of deviation between surface regions.
Hausdorff Distance

- approximate mesh neighborhood
- original mesh
Deviation Space

- most deviation is orthogonal to the surface
- uniform component required by curved surfaces
- deviation value now depends on viewing angle:

\[ D_{\hat{n}}(\mu, \delta) = \max(\mu, \delta \|\hat{n} \times \vec{v}\|) \]
Screen-space Geometric Error

✧ Residual error vectors: \( E = \{ e_i \} \)

✧ Determine the ratio:

\[
\frac{\delta_v}{\mu_v} = \max_{e_i \in E} (e_i \cdot \hat{n}_v) / \max_{e_i \in E} \| e_i \times \hat{n}_v \| 
\]

✧ Find the smallest \( D(\hat{n}_v, \delta_v, \mu_v) \) that bounds \( E \).

✧ Check to see if the screen space projection of \( D(\hat{n}_v, \delta_v, \mu_v) \) exceeds \( \tau \); if it does, refine.
Results
Incremental Selective Refinement Algorithm

- Basic idea: traverse the list of active vertices before rendering each frame, and for each vertex, either leave it as is, split it, or collapse it.

```
procedure adapt_refinement()
    for each v ∈ V
        if v.vt and qrefine(v)
            force_vsplit(v)
        else if v.parent and ecol.legal(v.parent) and
            not qrefine(v.parent)
            ecol(v.parent) // (and reconsider some vertices)

procedure force_vsplit(v') {
    stack ← v'
    while v ← stack.top()
        if v.vt and v.fl ∈ F
            stack.pop() // v was split earlier in the loop
        else if v ∉ V
            stack.push(v.parent)
        else if vsplit.legal(v)
            stack.pop()
            vsplit(v) // (placing v.vt and v.vu next in list V)
    else for i ∈ {0...3}
        if v.fn[i] ∉ F
            // force vsplit that creates face v.fn[i]
            stack.push(v.fn[i].vertices[0].parent)
```
Time Complexity

- Worst Case: $O(|V^a| + |V^b|)$
- $M^a \rightarrow M^b$ might be $M^a \rightarrow M^0 \rightarrow M^b$
Frame Rate Regulation

- Variability in frame rates caused by varying number of active faces
- Maintain a constant number of active faces using a simple feedback mechanism that regulates the screen-space tolerance.

\[ \tau_t = \tau_{t-1}(|F_{t-1}|/m) \]
Results

Figure 9: Measurements in flythrough for constant $\tau = 0.25\%$ (1.5 pixels in $600^2$ window). From top: number of faces in thousands, $\tau$ in pixels, frame times and adapt.refinement times in seconds.

Figure 10: Same but with regulation to maintain $|F| \approx 9000$. ($\tau$ is never allowed below 0.5 pixels.)
Amortization

Figure 9: Measurements in flythrough for constant $\tau = 0.25\%$ (1.5 pixels in 600° window). From top: number of faces in thousands, $\tau$ in pixels, frame times and adapt.refinement times in seconds.

Figure 10: Same but with regulation to maintain $|F| \approx 9000$. ($\tau$ is never allowed below 0.5 pixels.)

Table 2: CPU utilization (on a 150MHz MIPS R4400).

<table>
<thead>
<tr>
<th>procedure</th>
<th>% of frame time</th>
<th>cycles/call</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td></td>
<td></td>
</tr>
<tr>
<td>adapt.refinement (vsplit)</td>
<td>14 %</td>
<td>2200</td>
</tr>
<tr>
<td></td>
<td>(0 %)</td>
<td>4000</td>
</tr>
<tr>
<td></td>
<td>(1 %)</td>
<td>230</td>
</tr>
<tr>
<td></td>
<td>(4 %)</td>
<td></td>
</tr>
<tr>
<td>render (tstrip/face)</td>
<td>26 %</td>
<td>600</td>
</tr>
<tr>
<td>GL library</td>
<td>19 %</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>System</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS + graphics</td>
<td>21 %</td>
<td></td>
</tr>
<tr>
<td>CPU idle</td>
<td>20 %</td>
<td></td>
</tr>
</tbody>
</table>
Results

(a) Top view ($\tau = 0.0\%$; 33,119 faces)
(b) Top and regular views ($\tau = 0.33\%$; 10,013 faces)

(a) Original $M$ (19,800 faces)
(b) Front view and (c) Top view ($\tau = 0.075\%$; 1,422 faces)
Results

Figure 14: View-dependent refinement ($\tau = 0.15\%$; 1,782 faces) of a truncated PM representation (10,000 faces in $M$) created from a tessellated parametric surface (25,440 faces). Interactive frame rate near this viewpoint is 14.7 frames/sec, versus 6.8 frames/sec using $M$.

Figure 15: Two view-dependent refinements of a general mesh $\hat{M}$ using view frustums highlighted in orange and with $\tau$ set to 0.6%.
Results

Figure 16: View-dependent refinement. Interactive frame rate near this viewpoint is 6.7 frames/sec, versus 1.9 frames/sec using $\tilde{M}$. 
Progressive Simplicial Complexes

**Motivation:**
Abstract Simplicial Complexes

- Let $K$ be an abstract simplicial complex.
- Non-empty subsets of each element in $K$ are also guaranteed to be in $K$.
  - Given $\{j, k, l\}$ in $K$, $\{j, k\}$, $\{k, l\}$, $\{j, l\}$, $\{j\}$, $\{k\}$, $\{l\}$ also in $K$.
- The faces of a simplex $s$ is the set of non-empty subsets of $s$. 

\begin{center}
\includegraphics[width=\textwidth]{simplicial_complex_diagram}
\end{center}
Abstract Simplicial Complexes

✦ A simplex containing exactly $d+1$ vertices has dimension $d$ and is a $d$-simplex. (e.g. triangles are 2-simplices.)

✦ The $\text{star}(s)$ is the set of simplices of which $s$ is a face.
Abstract Simplicial Complexes

- The children of a $d$-simplex => $(d-1)$-simplices of $s$’s faces.
- The parents of a $d$-simplex => $(d+1)$-simplices of $\text{star}(s)$.
- Simplex with no parents is principal simplex.
- Simplex with exactly one parent is a boundary simplex.
PSC Representation

- A triangulated model as a tuple $M=(K,V,D,A)$.
  - $K$ is represented as an incidence graph of the simplices.

```c
struct Simplex {
    int dim;       // 0=vertex, 1=edge, 2=triangle, ...
    int id;
    Simplex* children[MAXDIM+1];  // [0..dim]
    List<Simplex*> parents;
};
```

- To render the model, draw only the principal simplices of $K$. 
PSC Representation

- The discrete attributes (D) associate a material identifier with each simplex
- Material identifiers contain *smoothing groups*, so creases form between pairs of adjacent triangles in different groups.
PSC Representation

- Area parameters (A) are associated with 0- and 1-simplices.
- Points and edges rendered as either a sphere or cylinder, a 2D point or line, or not rendered at all depending on screen pixel radius.
PSC Representation

Figure 3: Example of a PSC representation. The image captions indicate the number of principal \{0, 1, 2\}-simplices respectively and the number of connected components (in parenthesis).
Level-of-Detail Sequence

- Determine a sequence of `vunify` transformations simplifying the original mesh down to a single vertex.
- Reversing the transformations, we arrive at the record of generalized vertex splits which constitutes a PSC.
- For better approximations of lower complexity meshes, sum of areas is kept constant within each manifold component during simplification.
Vertex unification transformation

\[ v_{uni}fy(a_i, b_i, midp_i) : M^{i+1} \rightarrow M^i \]

- takes an arbitrary pair of vertices \( \{a_i\}, \{b_i\} \) and merges them into a single vertex \( \{a_i\} \).

- References to \( \{b_i\} \) in all simplices of \( K \) are replaced by references to \( \{a_i\} \).
### Generalized Vertex Split

<table>
<thead>
<tr>
<th>original simplex in $K^i$</th>
<th>corresponding simplices in $K^{i+1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>code (1)</td>
</tr>
<tr>
<td>0-dim ${a_i}$</td>
<td>undefined</td>
</tr>
<tr>
<td>1-dim</td>
<td>${b_i}$</td>
</tr>
<tr>
<td>2-dim ${a_i}$</td>
<td>${a_i}$</td>
</tr>
</tbody>
</table>
PSC Construction

- Algorithm almost identical to the one in original PM paper.
- Minor changes to the cost function.
- Put candidates pairs in a priority queue, ordered by ascending cost.
- Each iteration, merge vertex pair at the front, update costs of the remaining.
Candidate Vertex Pair Set

- A set of candidate vertex pairs is formed
  - set includes the 1-simplices of $K$
  - additional pairs that allow distinct connected components of $M$ to merge
- Use Delaunay triangulation to form these additional pairs
Selecting Vertex Unifications

\[ \Delta E = \Delta E_{dist} + \Delta E_{disc} + E_{\Delta area} + E_{fold}. \]

✦ Same as before:

\[ \Delta E_{dist} = E_{dist}(M^i) - E_{dist}(M^{i+1}) \]

✦ \( E_{dist}(M) \) = measure of geometric accuracy

✦ \( E_{dist}(M) \) calculated as the sum of the squared distances to \( M \) from a dense set of points sampled from the original model.
Selecting Vertex Unifications

\[
\Delta E = \Delta E_{\text{dist}} + \Delta E_{\text{disc}} + E_{\Delta \text{area}} + E_{\text{fold}}.
\]

- \( E_{\text{disc}}(M) = \) measure of geometric accuracy of discontinuities
- “Sharp simplices”:
  - boundary simplex (simplex with exactly 1 parent)
  - parents with different material identifiers
- Minimization preserves geometry of material boundaries, creases, and triangulation boundaries.
Selecting Vertex Unifications

\[ \Delta E = \Delta E_{\text{dist}} + \Delta E_{\text{disc}} + E_{\Delta \text{area}} + E_{\text{fold}}. \]

- \( E_{\Delta \text{area}} \) penalizes for surface stretching.
- \( E_{\text{fold}} \) penalizes surface folding to prevent model self-intersections.
Figure 9: For each column, the top row shows the original model and the bottom row shows one approximation in the PSC sequence. The image captions indicate the number of principal \(\{0, 1, 2\}\)-simplices respectively and the number of connected components (in parenthesis).
Discussion

✦ What does the PSC representation really contribute?
✦ There aren’t really any differences in the way the PMs and PSCs are created.
✦ PMs and PSCs have the same properties.
✦ The only thing different now is that we are treating a collection of objects as one object.
✦ Can’t we do that by allowing disconnected vertices to merge?