Efficient Map, Road, Terrain, Text and POI Rendering on OpenGL-ES

Tom McReynolds, NVIDIA
Mapping is Different

- Embedded platform: weak CPU with small cache, slow bus
- Need consistent latency, must be smooth *all the time*
- Similar to Vis-Sim apps
  - Constant frame rate, similar techniques
  - But lots of annotations: text, icons, landmarks, etc.
- Different “graphics culture”
Mapping Elements

- Scrolling landscape
- Terrain geometry, land use and roads
  - Terrain and roads have level-of-detail control
- Multiple icons/text/landscape features
  - Must be clear and uncluttered
- Scene realism features
  - Sky/fog, lighting, bump mapping
Strategy

- Use good graphics programming practice
- Performance tune application
- Solve mapping problems
  - Use “tricky” OpenGL-ES techniques
  - Stay within application constraints
Good OpenGL-ES Practice

• VBOs vs. vertex arrays
• Optimize rendering thread
  • Start rendering early in the frame
  • Maximize pipelining; keep input fifo full
• State sorting
  • Minimize state changes
• Primitive grouping
  • Combine primitives where possible, re-use verts, textures
Performance Tuning

• Benchmark to find bottlenecks, don’t assume you know!
  • Look for performance that varies as well
• Look for latency as well as bandwidth problems
  • Can wait for bus, disk, context switch, etc.
• Having some kernel and user tracing tools essential
Extensive Benchmarking

2D Performance Improvement in fps

Zoom Level

Baseline
R6 Baseline
08.03.07 Baseline
08.10.07 Baseline .2 DB
08.22.07 Baseline .2 DB
08.27.07 Baseline .2 DB
Tracing to find timing delays
Visualizing your Data: PerfHUD
Measuring Data
Mapping Requirements

- Roads and land usage
- POI collision detection
- Terrain scrolling
- 3D terrain generation
- Bump mapping
- Sky and fog
- Color correction
• High quality AA is expected
  • Better than what FSAA can routinely achieve
• Roads complex to draw
  • Fancy joins and ends
  • Road segments have multiple elements
• Straightforward implementation too slow
  • Low bus bandwidth from CPU to GPU
Road Techniques

• Use textures on tri-strips
  • Texture with alpha + alpha blend: high quality AA edges
  • Simplifies geometry: reduces CPU and bus bandwidth requirements
  • Works around slow wide lines on GPU
AA Roads using Textures

- Create “bullseye texture”
  - May need multiple resolutions to achieve good texel sampling at different zoom levels.

- “Stretch” texture over tri-strip road segments
  - S coords 0 at one end, 1 at other, .5 at line end transition.
  - T coords span width of line
AA Roads using Texture

Road detail textures, different resolutions (if needed)

Tri-strip with texture coordinates

Road texture stretched onto tri-strip
AA Roads using Texture (cont).

- Not limited to line ends
  - Can also do joins (curves) and intersections
  - Use finer tessellation of base strip to get smooth curves
- Discontinuous texture coordinates possible by doubling vertices
Land Use Features

- Same technique as roads
  - Expand to general polygons
  - Wrap polygons with textured tri-strips
- Alternatively you can surround polygons with AA lines
POI, Text Collision Detection

• POI = Points of Interest
  • Think road signs: food, gas, lodging, etc.
  • Usually round or square
  • 2D: a point on map, stays upright
  • 3D: billboard floating over point on terrain

• Text
  • City, county, country, names etc.
  • Sometimes aligned to stay upright
Roads, Land Use, Text and POIs
POI/Text Collision Detection

• Need to decide how many POIs to show
  • They shouldn’t overlap, or crowd each other: make map too busy or unreadable
  • More complex problem for 3D maps
• Need to keep POIs, rotated text from overlapping with upright text
  • Some apps require culling, not clipping
Applying POIs and Text

• Can blit POIs and text to map image
  • But this prevents fast updates
  • Corrupts base image, limiting reuse

• Better to add text, POIs as textured polygons
  • Rotation, scale invariant (within limits)

• Billboardling used to keep text, POIs upright
  • Can group billboardling rotations
Collision Detection Approaches

- Precompute geometry extents, overlaps with CPU
- If objects can clip each other, depth buffering and stenciling will work
  - Sort objects by priority; higher priority objects obscure lower priority ones
- Pixel picking methods to find overlap
  - Use colors as object ids
  - Needs fast readback, histogram
Terrain Scrolling

- Potentially large amount of bandwidth required
- Different techniques possible: best depends on hardware capabilities
  - Render speed
  - Blit Speed
  - Texturing
- Want to minimize CPU bandwidth, maximize GPU performance
Blitting

- Surface larger than visible area
  - Render a buffer around visible region
  - Render extra in direction of travel
  - Blit to scroll
- Optimal if there is not texturing h/w
- Uses a lot of video memory
- Blits may be slow, use up frame time
- Won’t handle rotations, scales
Blitting (cont)
Re-Render

• Surface the same size as visible area
• Re-render entire scene each frame
• No wasted video memory
• No copies, handles rotation, scaling
• Requires scene update, render update is very fast
  • H/W accelerated transforms
  • VBOs or other methods to minimize triangle bandwidth
Re-Render (cont)
Wrapped Textures

• Texture larger than visible region
  • Used texture coordinate wrapping
  • Render to region in front of direction of travel
  • Render with transformed texture coordinates to scroll
• No copy, handles rotation, some scaling
• Must have fast render-to-texture
Wrapped Textures
3D Terrain Generation

• Usually store height fields, create tessellated surfaces

• Multiple constraints
  • Must be fast with a slow CPU, bus
  • Data divided into tiles with different resolutions to ease loading, save space
    • Flat vs. mountainous terrain
    • Near vs. far tiles
  • Tiles must connect cleanly even when resolutions don’t match
Terrain Generation
3D Terrain Representation

- Sophisticated algorithms like SOAR
  - Good storage but high CPU load
- Simple regular grid
  - Fast but uses large amount of storage
- Simplified “quadtree” approach
  - Fast traversal to appropriate resolution tiles
  - Regular at each level, fast tessellation
Adding realism to Terrain

• The more the map matches windshield view the better, *but*:
  • Too busy = hard to understand
  • Too much detail, map goes out of date quickly

• The right amount of realism:
  • Easy for viewer to orient to map
  • Symbolic enough to quickly find useful details
More Detail not Always Better
Appropriate Terrain Realism

- Terrain shape
  - Simplified terrain ok, but can’t make distinct features unrecognizable!

- Lighting
  - Daytime: time of day (exact sun position)
  - Night: modified headlight view

- Sky, fogging
  - Adds realism that doesn’t distract
  - Simplifies view, eases terrain generation
Terrain rendering tricks

• Normal maps
  • Less computation for CPU
  • Per pixel lighting, high quality

• Fog maps
  • OpenGL-ES fog too primitive
  • Maps give fine control of fogging function
  • Can change fog color with height (haze)
  • Can combine: RGB = color A = fogging
Fog Maps

Horizontal plane

Horizontal distance of the vector to the camera plane

Height of the vertex

Height of the fragment

Fog map RGB

Fog map A

horizontal distance of the fragment to the camera plane
Terrain with Normal, Fog Maps

2D Map ($C_m$)

Normal Map ($C_n$)

Diffuse Light ($C_D$)

Pass 1 goes to RenTarget

$C_m \cdot \text{dot}(C_n, C_D) = C_{\text{MapShaded}}$

Pass 2 goes to BackBuffer

Terrain Diffuse Color ($C_{\text{Terrain}}$)

Fog map A

Fog map RGB

$V$

$U$

$C_{\text{final}}$

$\text{Fog (}C_{\text{Fog}})$

$\text{Terrain Diffuse Color (}C_{\text{Terrain}})$
Color Correction

• Content designers choose specific colors
  • Especially route colors
• They don’t like it if your shading changes their colors!
• Approaches
  • Tone mapping: adjust colors to “undo” shading effect
  • Use stencil to mask out shading
Going Forward

• OpenGL-ES 1.X (fixed functionality) can do a lot
• But there’s room for improvement
  • Better map quality
  • Better performance
• Newer maps need more of both
  • Higher resolution displays
  • More detail (with on-line updates)
  • Driver viewpoint
Areas to Improve: Terrain

• Displacement mapping
• Use height field and flat grids as inputs
  • Grids can be 2D and low precision coords
• Height field stored as mipmap
  • Grid vertex + ds/dx, dt/dy used to mipmap to height value
• Fragment shader used for lighting
  • Compute normal from height map
Going Forward: Billboardding

- Vertex shader can do billboard computations on GPU
- Can batch multiple billboards into single draw call + uniform data
- Can also adjust billboard height based on height field stored as a texture
- Billboard position/orientation info reduced to an 2D point
Areas to Improve: Buildings

• Minimize geometry needed to render buildings
• Assume simplified buildings: floorplan + constant height
  • Reasonable for non-landmark buildings
• Send convex outlines of buildings
  • Can build arbitrary buildings from these
• Render as fans for tops
• Use geometry shader to build walls
Areas to Improve: Buildings (cont)

- Geometry shader to build walls
  - Send floorplans twice
  - Use primitive restart to separate array into building pieces
  - Geometry shader offsets per attribute height value, stitches quad array
  - Wall id attribute can be used to choose wall texture, etc.
- Facet normals computed from quads
Questions?