Tessellation in Viva Piñata

ACM SIGGRAPH August 2007

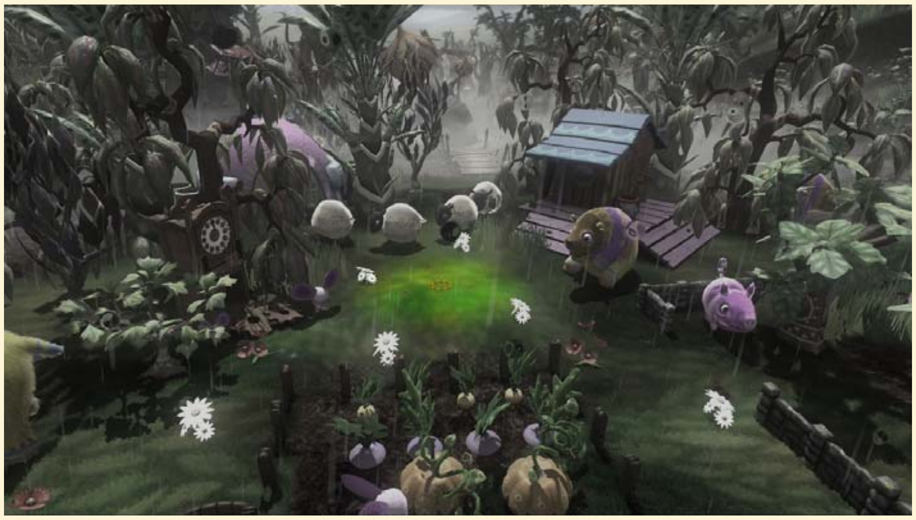
Michael Boulton
Senior Software Engineer
Rare / MGS

mboulton@microsoft.com

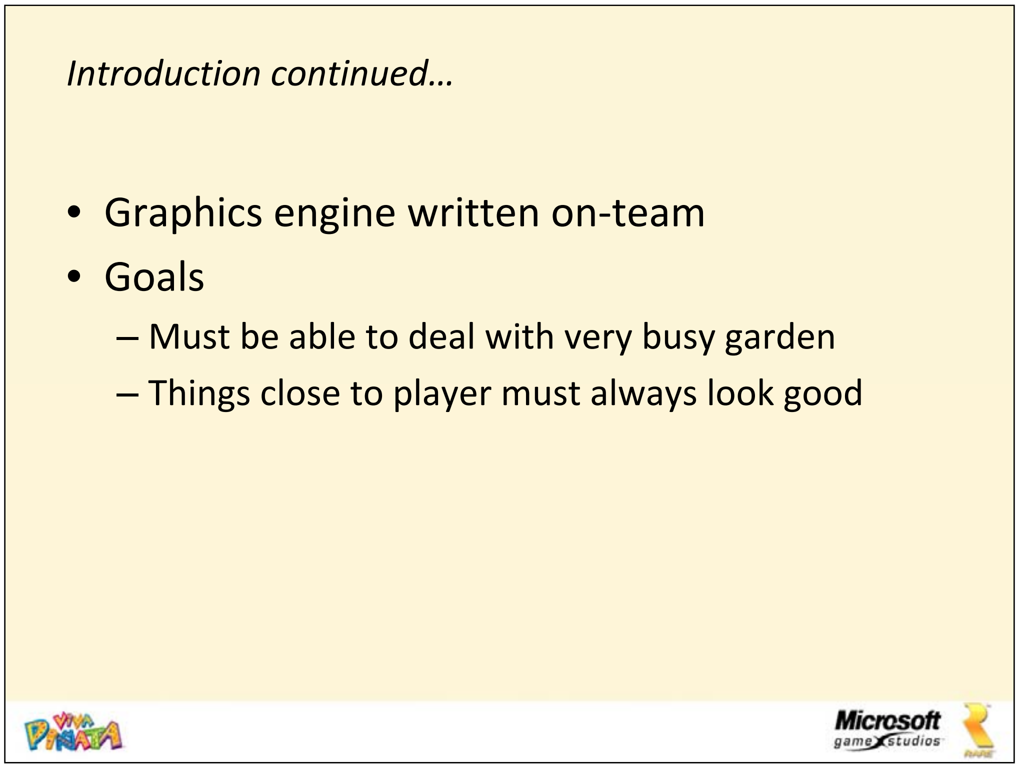
Introduction

- Viva Piñata released for the XBOX360, Nov 06

Although the game is cartoon-like, there is a formidable amount of GPU processing going on – Unified shadowing, lots of volumetric rendering, large draw distances and very expensive (microcode) shaders. This places a large emphasis on complexity management, which in our case involved level of detail for geometry and shaders, and tessellation.



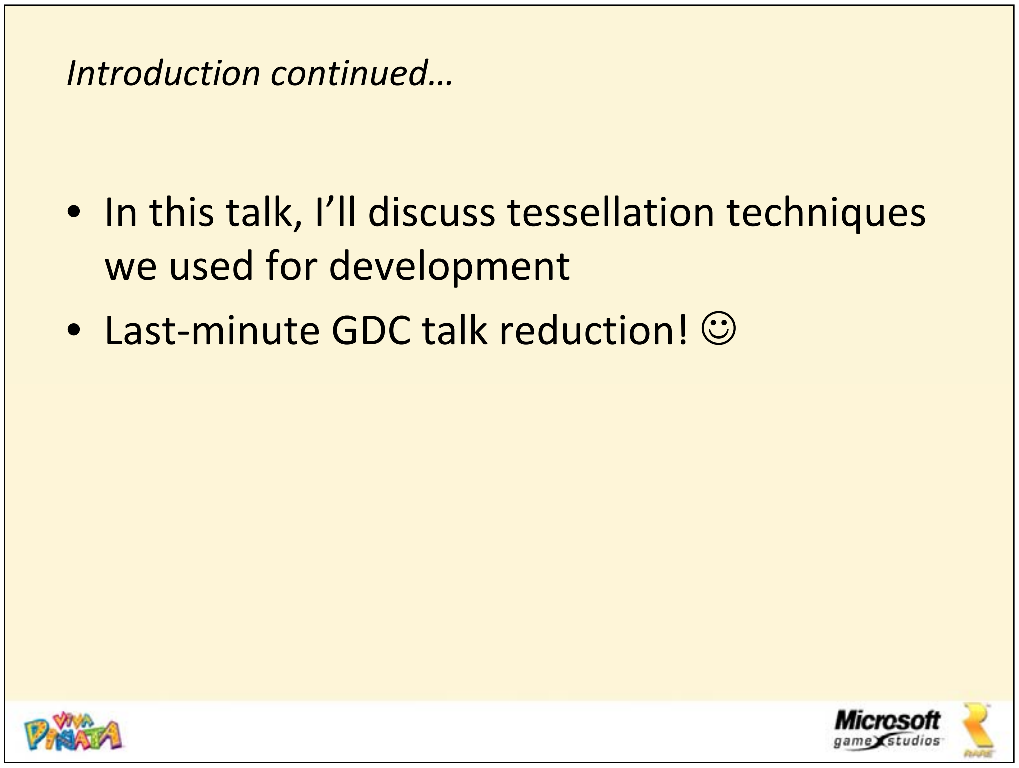
This screenshot gives an impression of just how detailed a single view can be, and highlights the potential complexity variation.



Introduction continued…

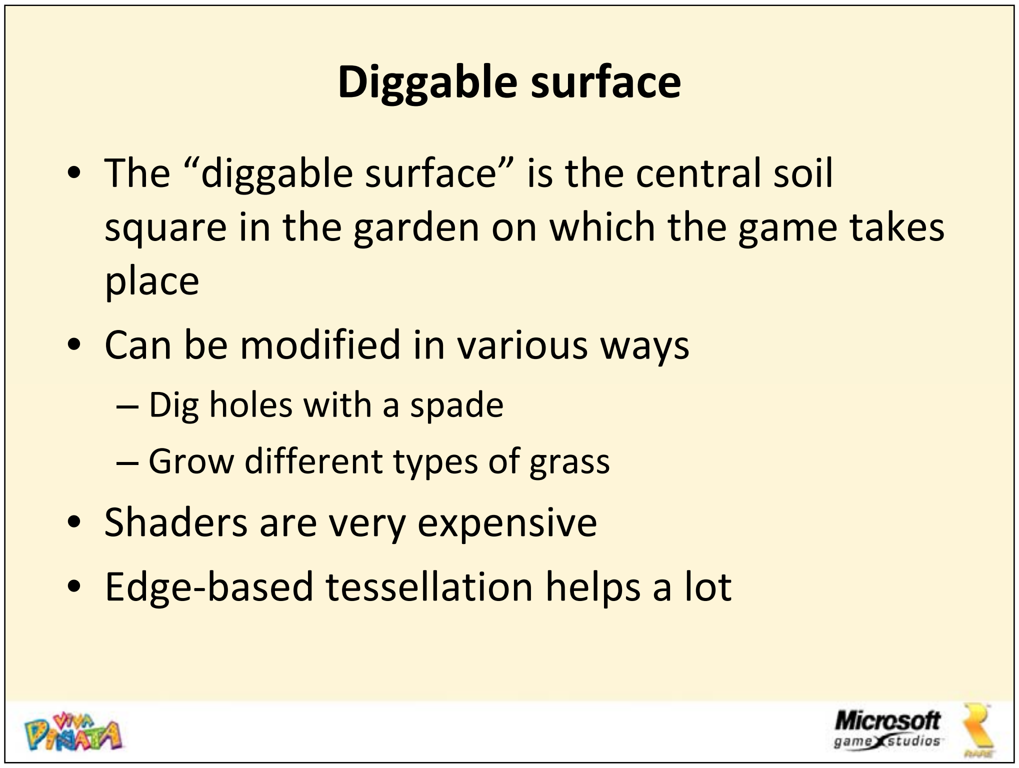
- Graphics engine written on-team
- Goals
  - – Must be able to deal with very busy garden
  - – Things close to player must always look good

The graphics engine was written on-team specifically for Piñata. This increased the burden of support, but gave us the ability to perform specific optimisations. This was of particular importance for Piñata because of the potential scene complexity, and the requirement that the game should always run at a steady frame rate.



Introduction continued…

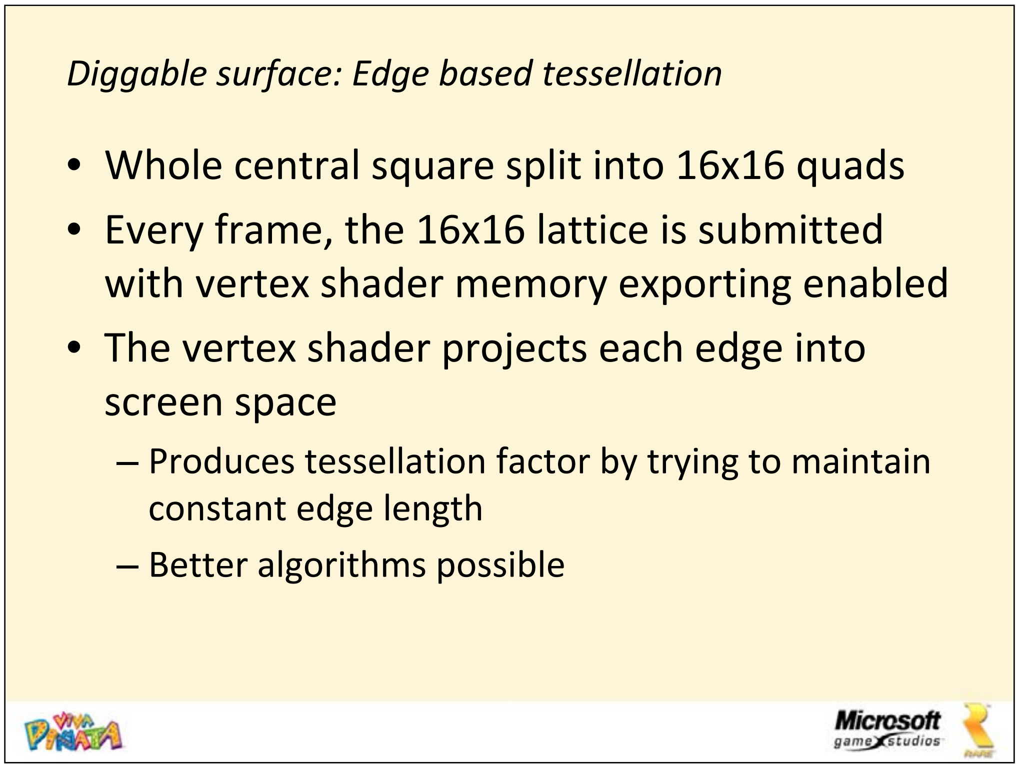
- In this talk, I’ll discuss tessellation techniques we used for development
- Last-minute GDC talk reduction! ☺



Diggable surface

- The “diggable surface” is the central soil square in the garden on which the game takes place
- Can be modified in various ways
  – Dig holes with a spade
  – Grow different types of grass
- Shaders are very expensive
- Edge-based tessellation helps a lot

The “diggable surface” is a good candidate for edge-based displacement because it’s mainly side-on to the camera. This would generate many small triangles if we used an un-tessellated mesh, which in turn would significantly increase the pixel work performed by the GPU.



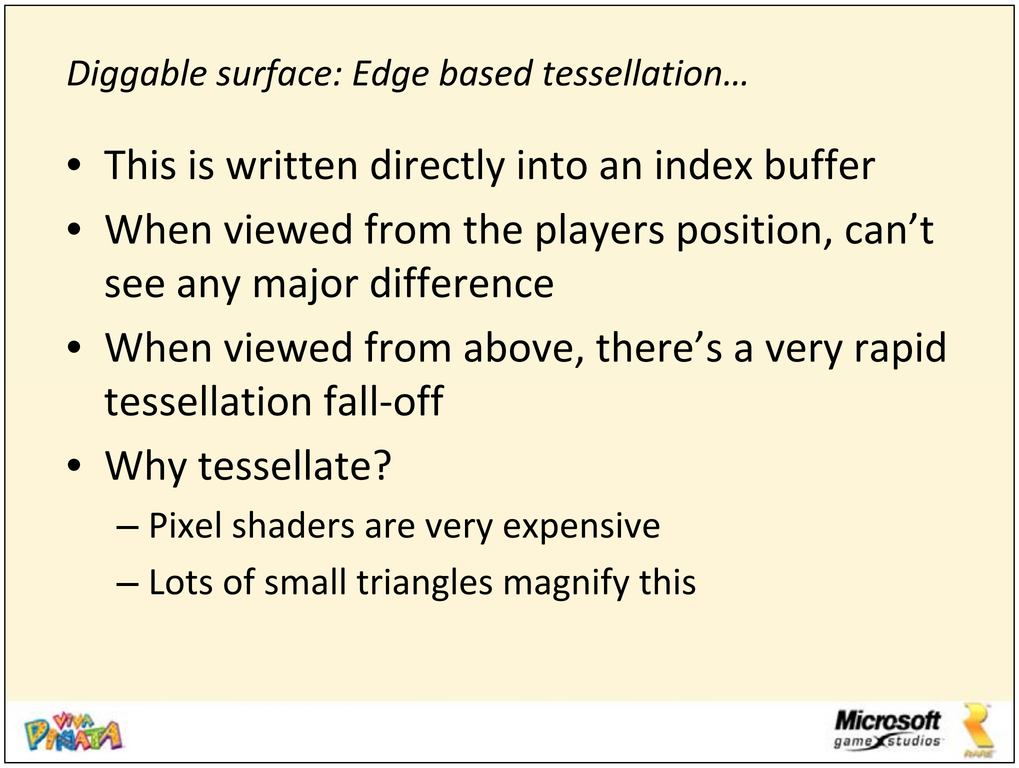
Diggable surface: Edge based tessellation

- Whole central square split into 16x16 quads
- Every frame, the 16x16 lattice is submitted with vertex shader memory exporting enabled
- The vertex shader projects each edge into screen space
  - Produces tessellation factor by trying to maintain constant edge length
  - Better algorithms possible

Each edge of the 16x16 quad lattice is projected into screen space, and the number of pixels along the projected edge is calculated. We calculate the required edge tessellation factor by trying to keep each tessellated edge component as close to fifty pixels in length as possible (so for instance if the projected quad edge had a length of 150 pixels, we would calculate a tessellation factor of 3).

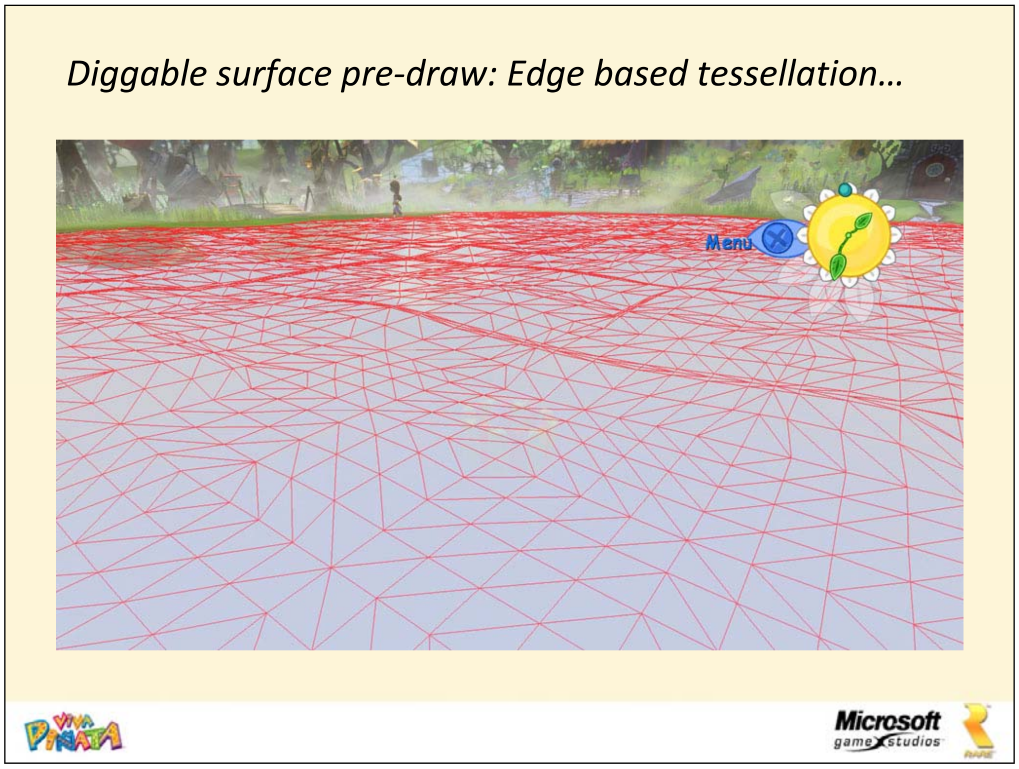
Each quad in the lattice is tessellated at least once so that the general character of the surface can be maintained (this prevents holes from completely disappearing).

Better algorithms – could calculate the projected area of each triangle, and work out edge factors from that.

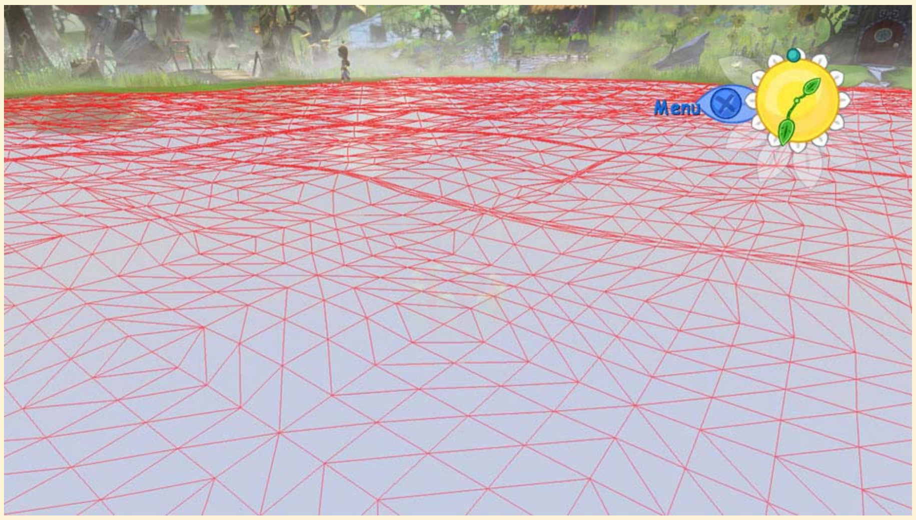


Diggable surface: Edge based tessellation…

- This is written directly into an index buffer
- When viewed from the players position, can't see any major difference
- When viewed from above, there's a very rapid tessellation fall-off
- Why tessellate?
  - Pixel shaders are very expensive
  - Lots of small triangles magnify this

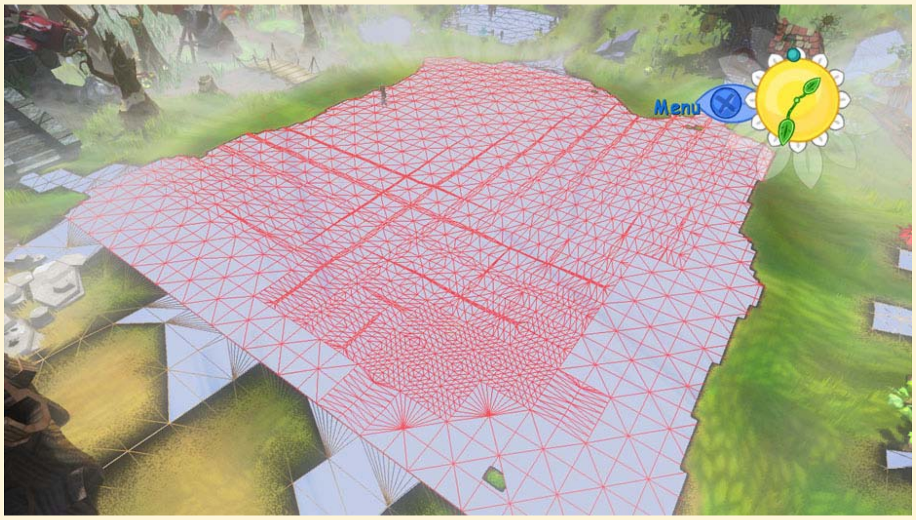


Diggable surface pre-draw: Edge based tessellation…



This is the view from the player position. Although tessellation is being performed, an acceptable density of triangles is still maintained.

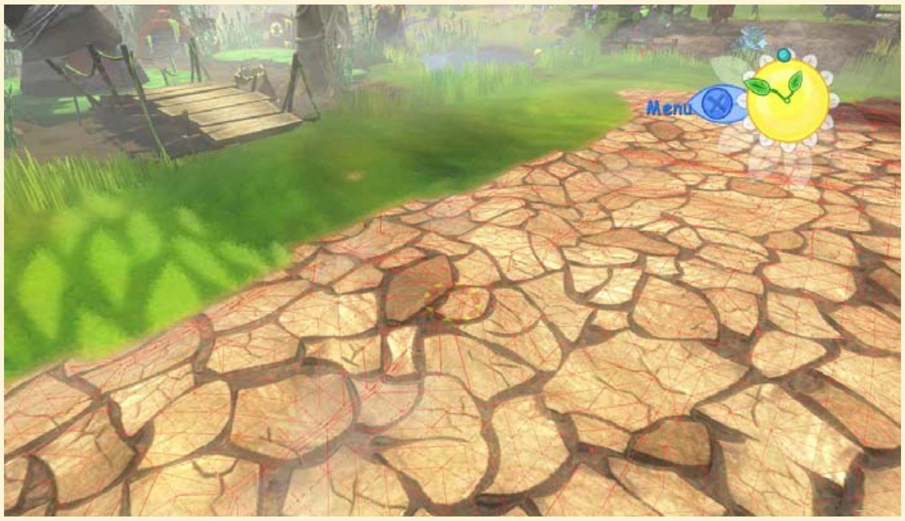
Diggable surface: Edge based tessellation…



Here, we have frozen the tessellation factors calculated for the previous image so that the fall-off can be observed from above. Notice how rapidly the tessellation factors drop down to the minimum as a function of distance from the viewer. Full tessellation is only used locally around the player.

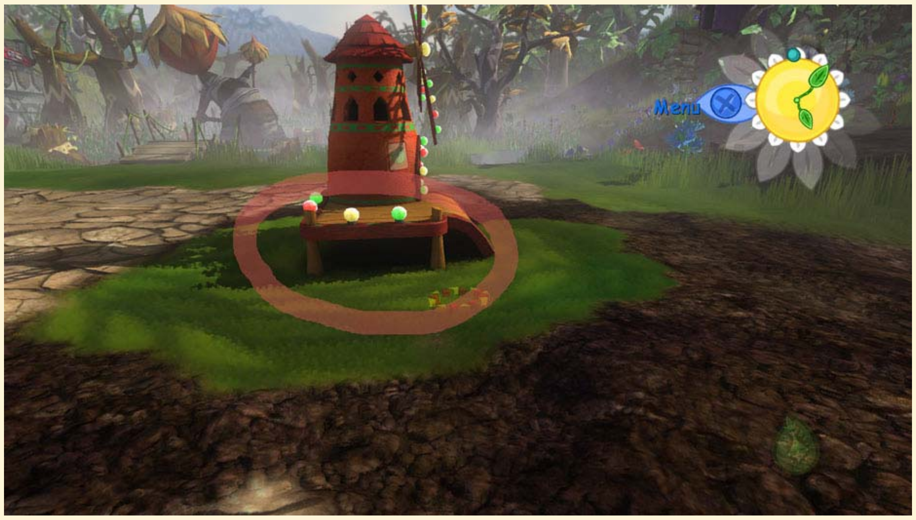
Also, notice that quads within the lattice that are entirely off-screen have their tessellation values set to the minimum in the microcode vertex export shader. This increases performance.

Diggable surface continued…



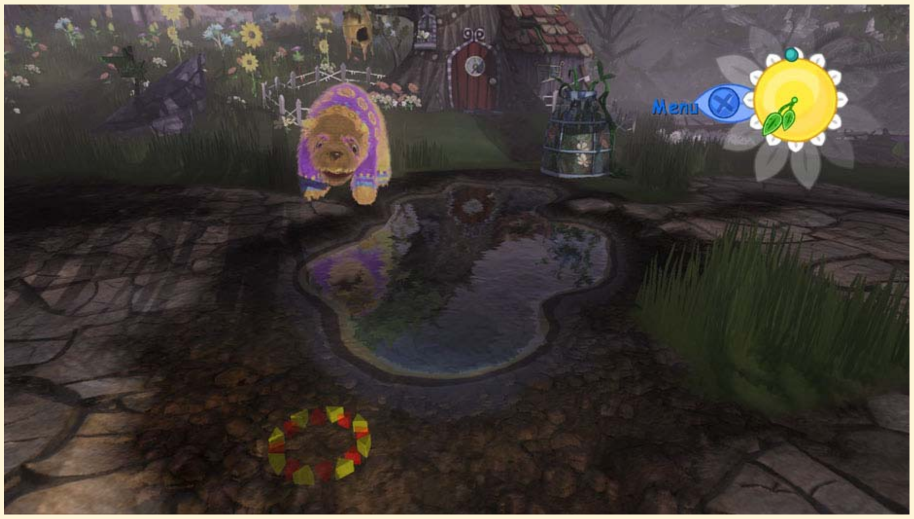
This video shows the tessellation in action (shown as overlaid red wireframe).

Diggable surface: Contact occlusion



One thing to be aware of if that many attributes can no longer be interpolated across vertices, since the changing tessellation factors can cause unacceptable swimming artefacts. This problem was addressed in Piñata by using a separate texture which encoded attribute values in a regular pattern, and was fetched within the vertex and pixel shaders (as demonstrated above with the grass and contact occlusion).

Diggable surface: Ponds



To avoid artefacts around pond banks, filtering was used in the vertex shader. When transitioning to the sediment texture layer, the height value was read directly from the attribute texture.

The vertex kill operator could also be used within a tessellated primitive, and was used to remove ground under grass (and vice versa).

Q & A

- Thanks for your time
- Any questions?
- Also, feel free to email
  - mboulton@microsoft.com