SHADOWGUN

Rendering techniques and optimization challenges

Petr Smilek / MadFinger Games
Renaldas Zioma / Unity Technologies @__ReJ__

UNITE 11 | Unity Developer Conference
San Francisco 28-30 September

MadFingerGames

Samurai, Samurai II
SHADOWGUN started February 2011
Team grew from 4 to 10

Thursday, September 29, 2011
About MadFinger Games

About 

- 3rd Person Sci-Fi Shooter
- Sci-Fi setting requires lot of eye-candy
  - cheap eye-candy ;)
- Large indoor & outdoor environments

Thursday, September 29, 2011

Shadowgun is 3rd person sci-fi shooter game developed on Unity engine

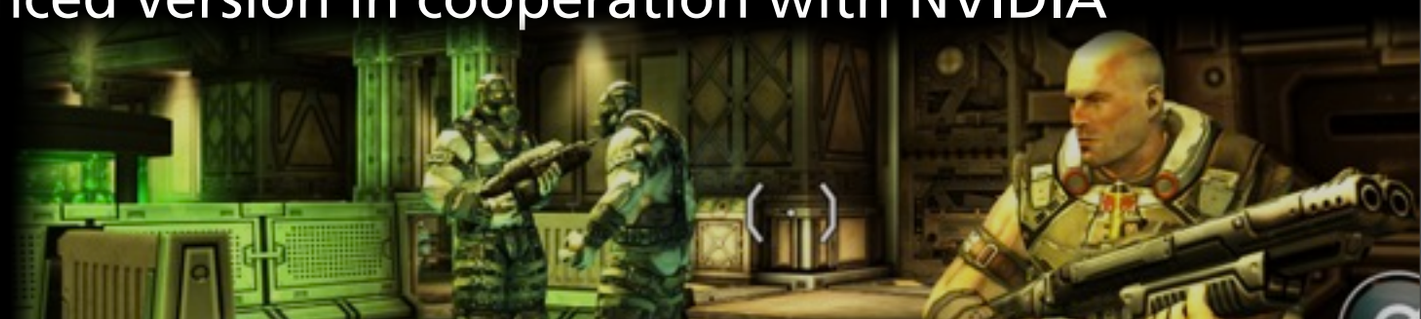
Sci-fi setting requires lot of (cheap ;)) eye-candy visuals \ effects to make game look attractive

Action takes place large indoor / outdoor (combined) environments

Trailer

Target Hardware

- iPad2
- Tegra2
- iPhone4, iPad, iPhone 3GS
- Coming Tegra3
  - Enhanced version in cooperation with NVIDIA



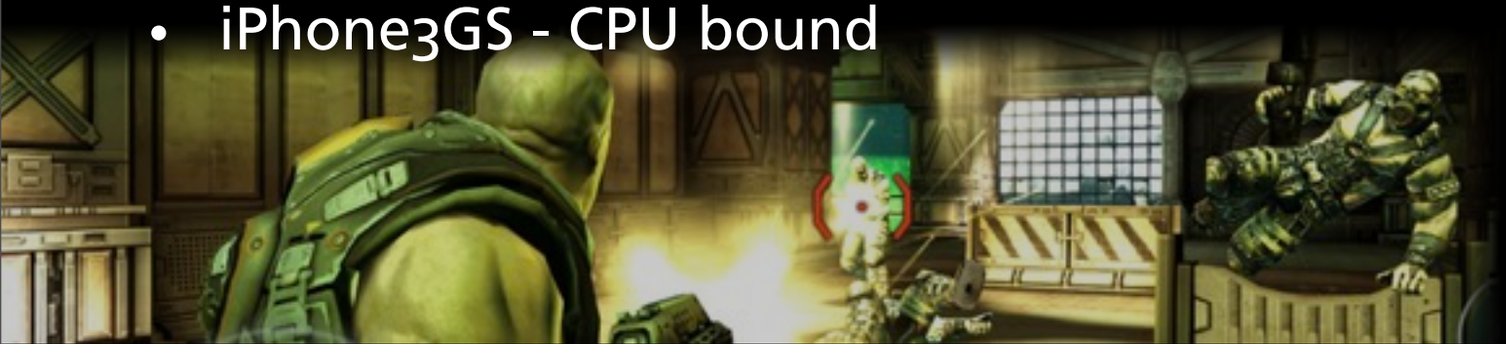
Thursday, September 29, 2011

Game is targeted for iOS devices (iPhone 3GS and higher, iPad) and for Tegra2 class Android devices. In cooperation with Nvidia we also work on enhanced Tegra3 version.

While those devices are very powerful with respect to their size / energy consumption, they are far away from desktop machines. To achieve at least similar visual quality as the games we can see on desktop machines / consoles, we must utilize lots of tricks (and cheat) to get the job done.

Hardware Challenges

- iPad2 - CPU bound
  - because GPU is very fast!
  - we utilize ~20% of 2 cores so far
- Tegra2, iPad, iPhone4 - GPU bound
  - lots of pixels: 1366x600 .. 960x640
- iPhone3GS - CPU bound



Hardware Challenges: GPU

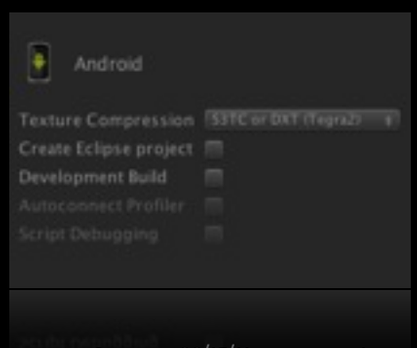
- Lots of tricks and optimizations!
- iPad2, 4xMSAA - 9ms on GPU (~100FPS)
  - 1024x768
  - we have lots of room for even better graphics here!
- iPhone4 - 30ms on GPU
  - 960x640
- Tegra2 - 30ms on GPU
  - 1366x600

Hardware Challenges: GPU/CPU

- Hand optimized shaders
  - Precision is very important
- Rendering order of opaque geometry
  - Tegra: big occluders first - front to back, rest by shader
  - iOS: sort by shader
- Improved dynamic geometry submission
- Coming in next Unity version

Hardware Challenges: Memory

- iPad, iPhone3GS, Android phones
  - under 128MB of free memory
- Just skip highest texture mipLevel
- Use DXT5 on Tegra




```
if (iPhoneSettings.generation == iPhoneGeneration.iPodTouch3Gen ||
    iPhoneSettings.generation == iPhoneGeneration.iPodTouch4Gen ||
    iPhoneSettings.generation == iPhoneGeneration.iPhone3GS ||
    iPhoneSettings.generation == iPhoneGeneration.iPad1Gen)
    QualitySettings.masterTextureLimit = 1;

if (iPhoneSettings.generation == iPhoneGeneration.iPad2Gen)
    QualitySettings.antiAliasing = 4;
```

Lighting and shading

- Crucial for overall look

- GI Lightmaps - static objects
- GI Light Probes - dynamic objects
  - Together provide solution for consistent high-quality lighting for every kind of object
  - Light Probes are stored as Spherical Harmonics

Thursday, September 29, 2011

"Unity Advanced Shading and Lighting Demo" video: http://www.youtube.com/watch?v=-LWZQXzUnUI
Lighting and shading is crucial for overall game look
Shadowgun utilizes GI based lightmapping together with spherical harmonics based lighting for dynamic objects
SH lighting is new technology incorporated into Unity engine. Together with GI based lightmaps it provides complete solution for consistent high-quality lighting for every kind of object.

Environment Lighting

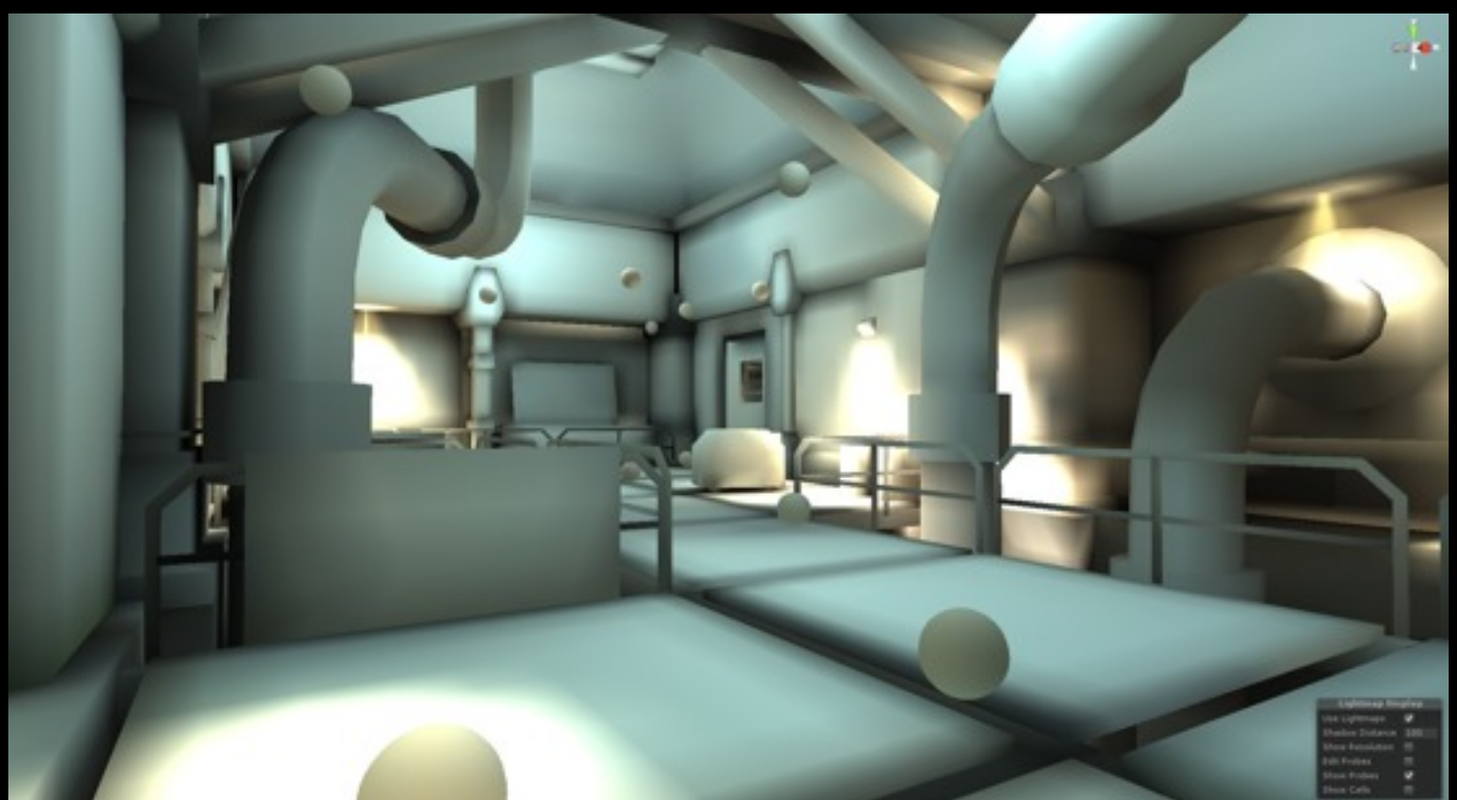
- Single 1024x1024 lightmap per level
  - uncompressed
  - we prefer low resolution, but perfectly smooth GI lighting
- ~500 light probes per level
  - every ~3.5 meters
  - such density gives perfect matching of dynamic objects and static environment

Thursday, September 29, 2011

Most levels fit into one 1024 x 1024 lightmap texture. We don't compress lightmaps because of introduction of nasty compression artifacts. We prefer lower frequency, but perfectly smooth GI lighting.

There are usually few hundreds (~500) of lightprobes in level, with ~3.5 meters spacing between them. This density of lightprobes guarantees perfect matching of dynamically lit objects with static environment.

GI = lightmaps + light probes



Crisp textures

- Anisotropic filtering
  - faster than mipMapBias
    - on SGX *tex2DBias* performs as dependent read
  - easy to scale performance
  - no aliasing!
  - portable

Fake specular highlights

- Proper specular highlights on lightmaps = expensive
  - need 2 or 3 times more memory
  - need to reconstruct at run-time
  - instead...
- “Virtual” specular-only light
  - attached to camera with some offset
  - per-vertex
  - masked with glossiness factor from Main texture’s alpha

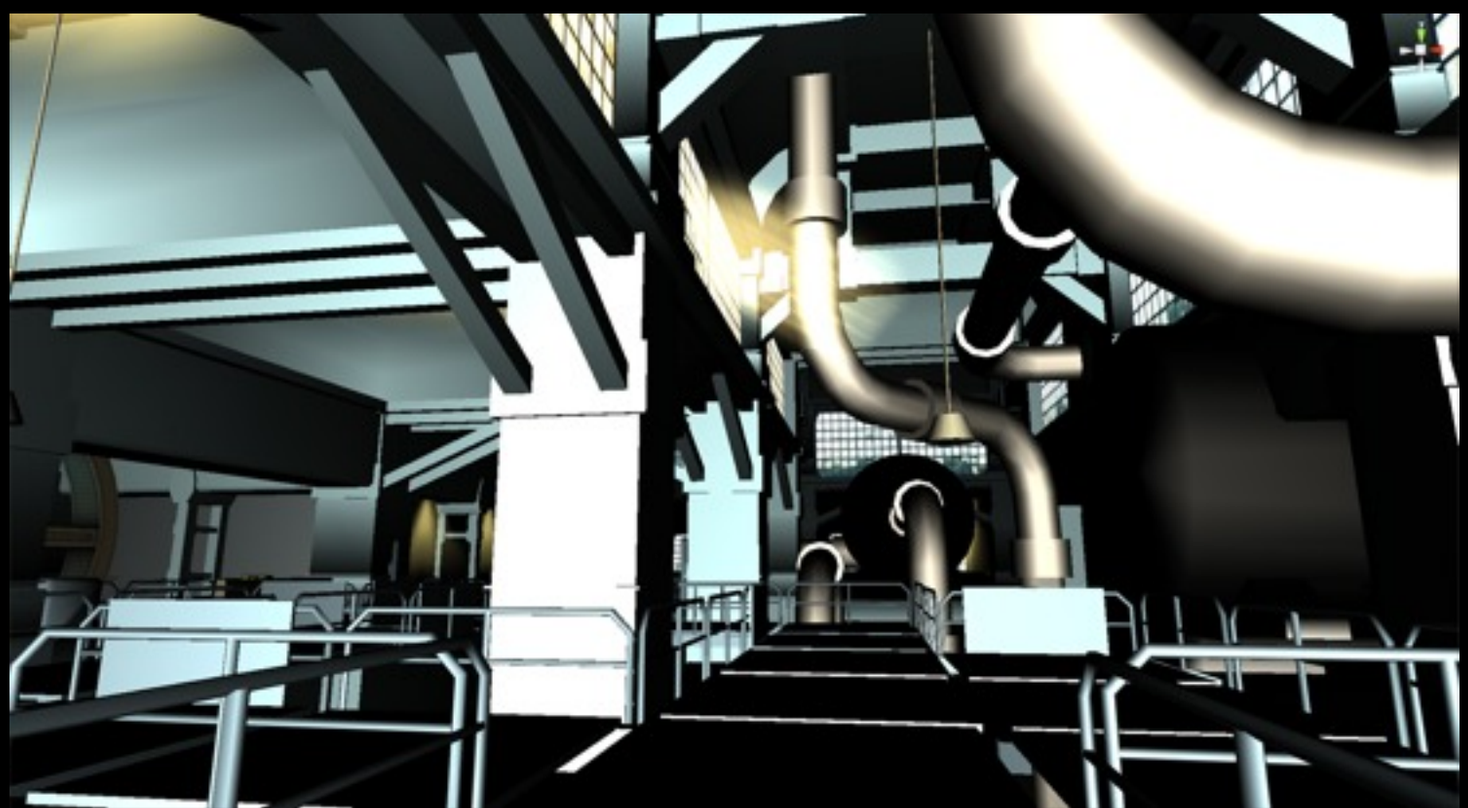
Thursday, September 29, 2011

Because of the usage of precomputed lighting it is not very strightforward to achieve reasonably looking specular highlights on lightmapped surfaces.

This would require lightmap to store more information than simple diffuse lighting. While this necessary information can be generated by Unity engine lightmapping process, we decided to go simpler way (mainly for performance reasons).

What we do is to put ‘virtual’ light source at camera position (with some offset) and generate per-vertex specular lighting using this light. Using this approach we get cheap specular highlights, which are not on correct positions, however “nobody” notices this as he get nice shiny looking surface

Fake specular highlights



“Volumetric” effects

“Volumetric” effects

“Volumetric” effects

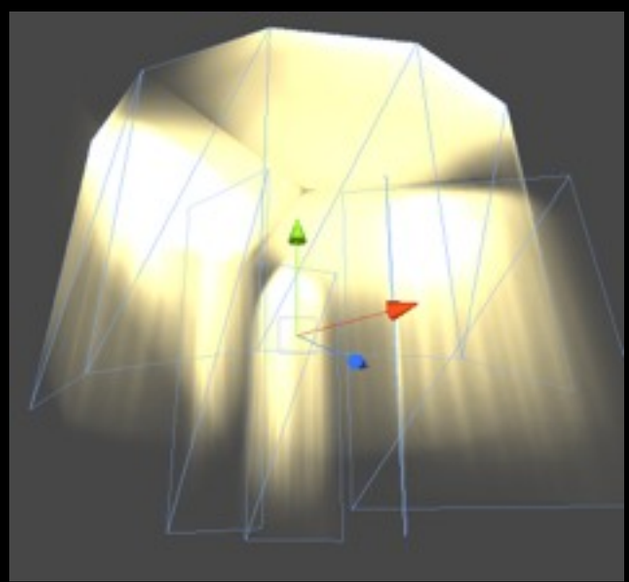
- Approximation of volumetric phenomena
  - Glows
  - Light Shafts
  - Fog Planes
  - Emissive Billboards
- Implemented as additively blended surfaces
- To achieve convincing effect need a lot of surfaces
  - Very expensive on mobile GPU
  - Heavily optimize / cheat

Thursday, September 29, 2011

Glows, light shafts, fog planes, emissive billboards are important visual property of good looking sci-fi setting
Usually considered to be approximation of volumetric phenomena and commonly implemented by rendering additively blended surfaces.
To achieve convincing effect you must render quite a lot of this surfaces. This, together with the fact that blending can be quite expensive on most GPUs forces us to heavily optimize / cheat.

Volumetric FX: optimizing fillrate (1)

- Simplest fragment shader
- In many cases do not even a texture
  - Procedurally calculate intensity per-vertex
  - If doesn't look smooth enough...
  - ... just use more vertices



Thursday, September 29, 2011

First step to optimize fragment processing of blended surfaces is obvious – use as simple fragment shader as possible.
For example in many cases fragment shader can just output color coming from vertex shader, so there is no need to even sample a texture (just use more vertices if linear interpolation of per-vertex evaluated function does not look smooth enough)

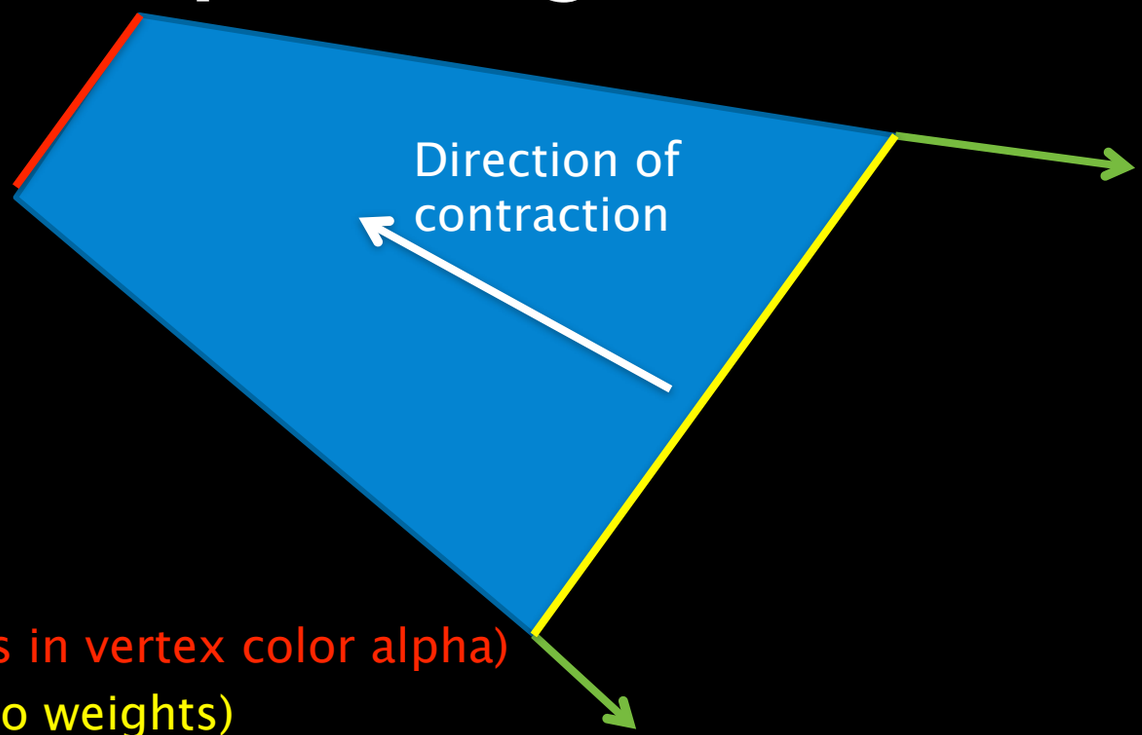
Volumetric FX: optimizing fillrate (2)

- Decrease transparency
  - when surface is too close to viewer
  - once transparency is closer to zero...
- Shrink surface
  - control shrink direction with normals
  - implemented in vertex shader

Thursday, September 29, 2011

Second step, which also improves visual quality is to do following:
Whenever surface is starting to be too close to viewer (and thus occupy lots of pixels on screen), start to decrease it transparency.
Once, the transparency is close to zero, start shrinking of surface in appropriate directions (controlled by manually placed vertex normals) until it shrinks to degenerated tris, which does not generate any pixels to rasterize. This process take place in vertex shader.
When done correctly, user does not notice the shrinking process (there are no visual pops). Because of viewer distance based fadeout, there are never ugly transparent faces intersecting camera near clipping plane.

Volumetric FX: optimizing fillrate (2)



Fixed edge (zero weights in vertex color alpha)

Shrinking edge (non-zero weights)

Normals (define shrink direction)

Volumetric FX: optimizing fillrate

Characters

Characters

- Textures
  - 1024 diffuse
  - 512 normals
    - not compressed
  - 128 light lookup
    - not compressed
- 2000 verts
- 25 bones

Character lighting

- Initially was looking for a fast way to do per-pixel specular
- Texture lookup to calculate diffuse + specular
  - RGB - diffuse
  - A - specular
- Very flexible + constant cost:
  - “Trilight”: Key, Back, Fill in one go
  - Wrap light for skins
  - Custom lobes for metallics
  - Energy conserving specular highlight
- Disadvantage: can be tricky on some GPUs
- Soon on Asset Store!

Character lighting

- GI lighting from Light Probes is evaluated per-vertex
  - Closest Light Probes are found (per object)
  - Combined into a single SH (per object)
  - Muzzle flash is added to SH (per object)
  - Lighting is computed by sampling from SH (per-vertex)
- Per-pixel lighting is modulated with GI from Light Probes
  - Not physically correct, but proved to be a good choice for Sci-Fi mood

Character soft-shadows

Character soft-shadows

- Approximate character by spheres
- Calculate analytic AO
  - how much hemisphere (sky) is occluded by sphere
  - min() func to combine AO contributions
- Evaluate in vertex shader
  - on a pre-tessellated planar geometry
- Math hard work done by Iñigo Quilez (Iq / RGBA)
  - see “Sphere Ambient Occlusion” for details

Thursday, September 29, 2011

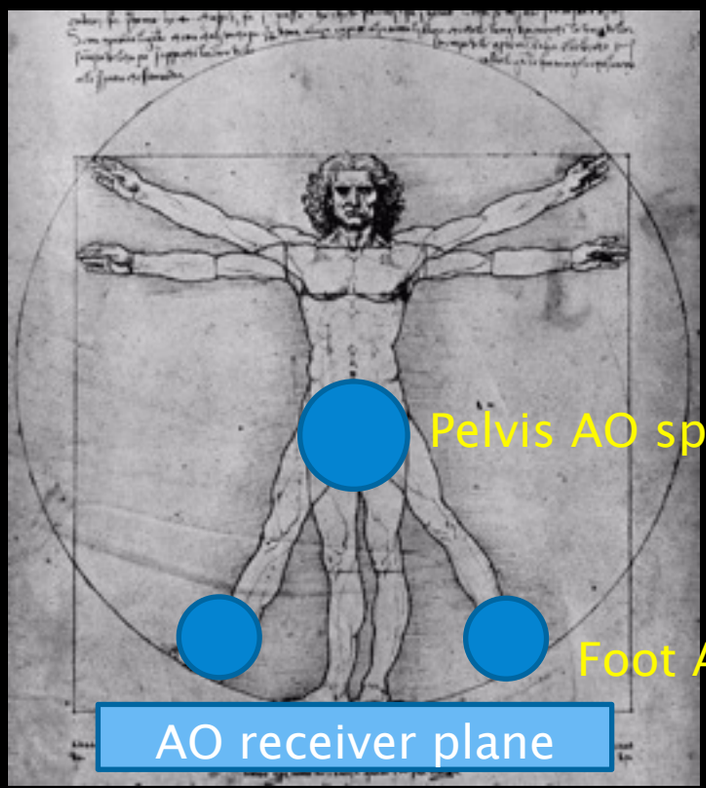
On most current mobile platforms, where GPU time is precious resource, we cannot usually afford nothing more but simple blob based character shadows

We are however mostly limited in fragment shader, so by putting more workload to vertex unit, we can do much better (still blob based) shadows.

Imagine that you approximate character by spheres (we use 3 spheres for Shadowgun – left foot, right foot, pelvis) and than calculate analytic AO (Ambient Occlusion) between spheres and plane in vertex shader. Use min-blending to combine AO contributions from spheres.

While derivation of analytical expression of plane vs sphere AO is not trivial, result is very simple to be evaluated in vertex shader. The actual math hard work was done by Iñigo Quilez (Iq / RGBA demogroup) – see his article “Sphere Ambient Occlusion” for details.

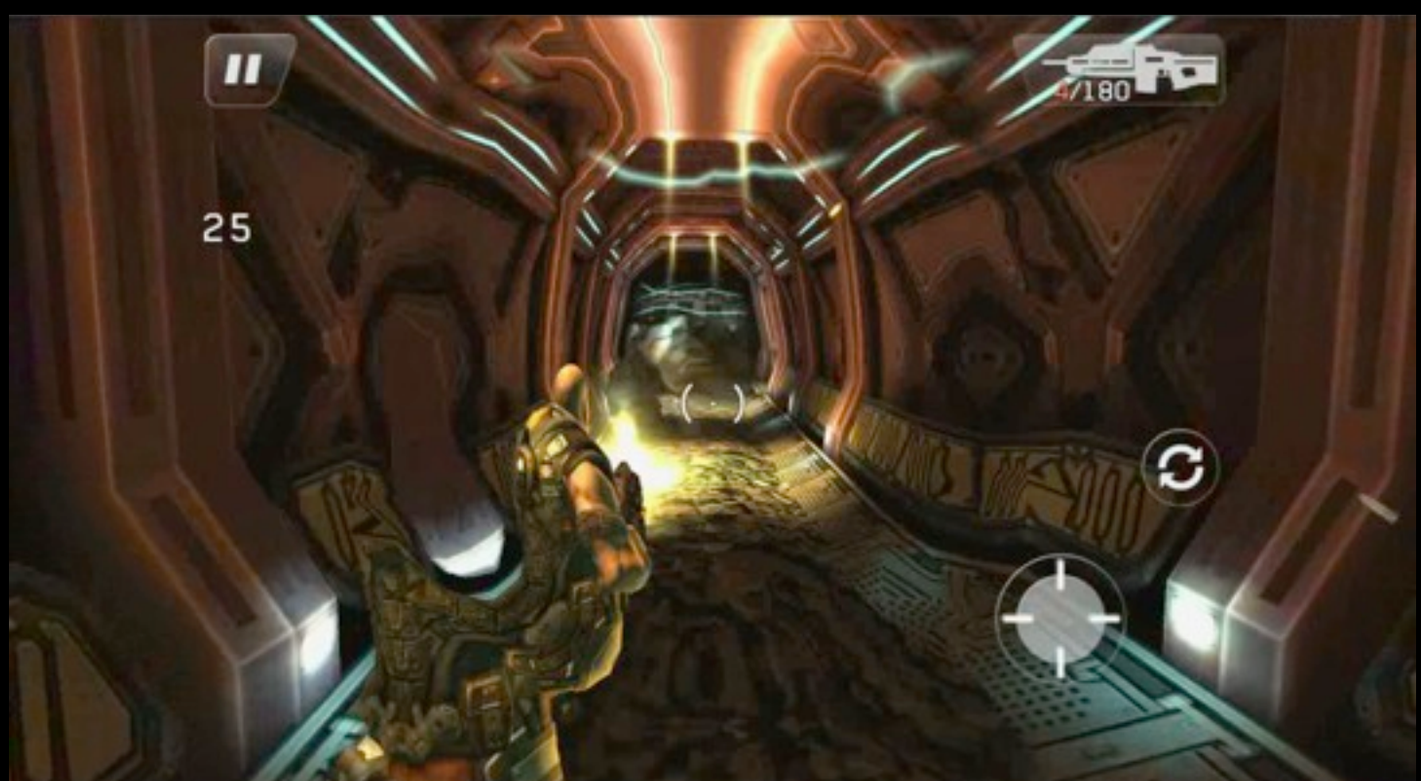
Character soft-shadows



Character soft-shadows

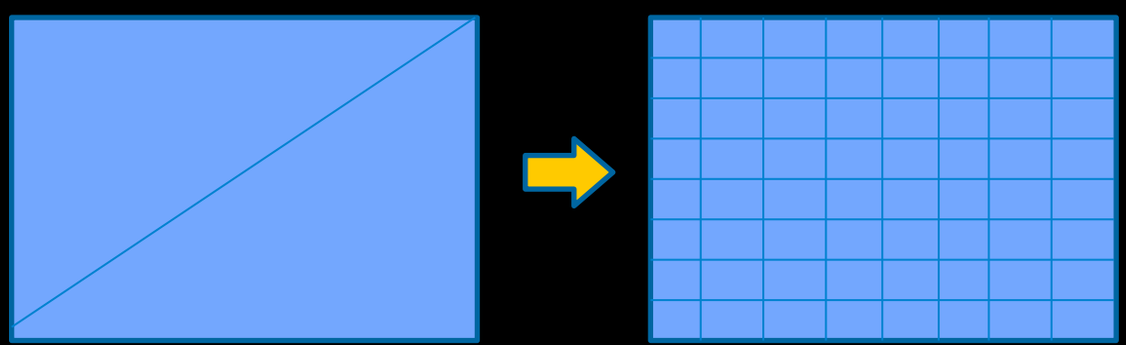
Character soft-shadows

Screen deformation FX



Screen deformation FX

- Full screen effects can be very expensive on mobile GPUs
- Calculate deformation on lower than screen resolution
  - Screen-space aligned grid
  - Deform in vertex shader



Classic PostFX

Vertex grid based PostFX

Thursday, September 29, 2011

Image deformation post-processing effects are very popular in modern games (hot air shimmering, deformation of image during explosions etc.)
Performing this transformation in pixel shader is too expensive to be used on today mobile GPUs.
However it is possible to perform same transformation in vertex shader with almost identical results in most cases.

Screen deformation FX

- Screen-space aligned grid
  - 30 x 25 vertices
- Distort UV per vertex
  - Fragment shader is very simple - only samples 1 texture
  - Distort up to 4 waves
- Calculate nice colorization
  - Based on projected explosion position in 2D
  - Emphasizes blast effect

Thursday, September 29, 2011

In vertex shader you calculate distorted UVs for each vertex, so fragment shader just performs simple source texture lookup with distorted UVs.
We calculate distortion from up to 4 waves during single pass to support multiple explosion effects on screen in one time.
For explosion effects you can also calculate nice colorization around position of explosion projected to 2D to further emphasize blast effect.

Interactive fluid surfaces

Interactive fluid surfaces

- Fluid surface is a 2.5D heightfield
- State is two 2D arrays
  - height
  - velocity
- Every frame
  - perform simulation step
  - reconstruct mesh (positions & normals) from heightfield

Thursday, September 29, 2011

Basic implementation of interactive fluid surface in 2.5D (heightfield) is very simple, but also very good looking.
You just keep two 2D arrays of floating point values (you can use fixed point if you wish), one holds current positions and another current velocities (both are scalar values)
At each frame you perform simulation step and reconstruct mesh from heightfield (positions and normals) and use it to render fluid surface

Interactive fluid surfaces

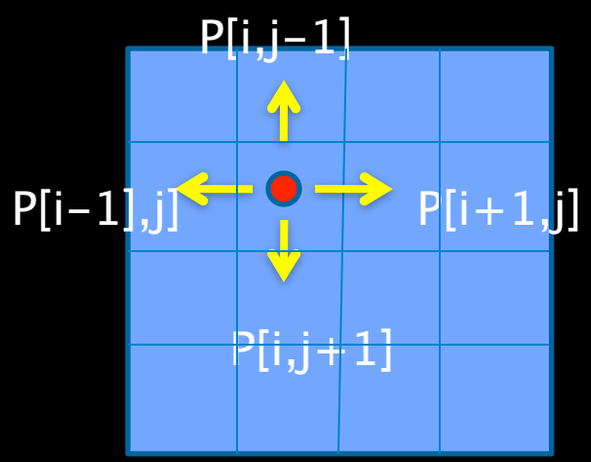
- Update velocities:

$$V_{t+1}[i,j] = V_t[i,j] + dT*c^2*(P_t[i-1,j] + P_t[i+1,j] + P_t[i,j-1] + P_t[i,j+1] - 4*P_t[i,j]) / h^2$$

  - dT … simulation step delta time
  - h … cell width
  - c … wave speed

- Update heights:

$$P_{t+1}[i,j] = P_t[i,j] + dT * V_{t+1}[i,j]$$



Interactive fluid surfaces

- Optimization tip:
  - add 1 pixel borders to avoid branches
- 128x128 fluid on Tegra3
  - C++ 0.9ms
  - Naive NEON 0.5ms

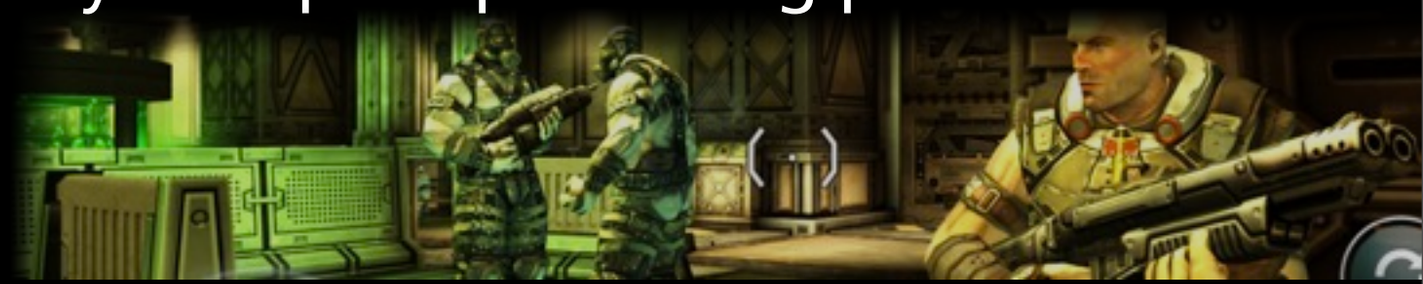
Thursday, September 29, 2011

Obvious optimization hint: When accessing 4-neighbourhood of processed “pixel”, don’t perform any clamping of (i,j) indices to avoid out of bounds access. Just use 1-pixel wide border which is not simulated at all and act like safeguard region.

Using straightforward C++ code simulation update of 128 x 128 fluid surface takes around 0.9 msec on Tegra3, so it is reasonably fast to be used without any fancy optimizations. Straightforward port to NEON SIMD assembly version takes about 0.5 msec on same device.

Shading water surface

- Blend between shallow & deep water colors
- Pre-rendered cubemap for reflection of the environment
- Fresnel term per-vertex
- Limited by GPU pixel processing performance



Thursday, September 29, 2011

Rendering realistic water surface requires complex shading, but unfortunately on current mobile GPUs we are quite limited in terms of pixels processing performance
In Shadowgun we calculate water color (simple blend between shallow \ deep water color) + Fresnel term in vertex shader
Pixel shader samples precalculated cubemap containing reflected environment and combines it with water color according to Fresnel term

References

- Iñigo Quilez : Sphere ambient occlusion
  http://www.iquilezles.org/www/articles/sphereao/sphereao.htm
- Matthias Müller : Real Time Fluids in Games
  http://www.cs.ubc.ca/~rbridson/fluidsimulation/GameFluids2007.pdf

Thursday, September 29, 2011

Rendering realistic water surface requires complex shading, but unfortunately on current mobile GPUs we are quite limited in terms of pixels processing performance
In Shadowgun we calculate water color (simple blend between shallow \ deep water color) + Fresnel term in vertex shader
Pixel shader samples precalculated cubemap containing reflected environment and combines it with water color according to Fresnel term

That's all – thanks for your attention ☺