Visualizing Large Image Datasets in 3D Using WebGL and Media Fragments
Overview

• Introduction
  • WebGL
  • Media Fragments
  • Virtual Texturing
• System Overview
  • Determining the working set
  • Page table generation
  • Page requests
• Server Side
• Results
• Conclusions
WebGL

- Spec. is v1.0 since 10/2/2011
- OpenGL ES 2.0 binding for HTML5 browsers
  - Shaders, render to texture, ...
- Extra limitations to ensure security
- Based on the canvas element
- Additional supporting classes and objects
  - Typed arrays: Int32Array, Float32Array, ...
- Browser support status:

<table>
<thead>
<tr>
<th>Browser</th>
<th>Chrome</th>
<th>Firefox</th>
<th>Safari</th>
<th>Opera</th>
<th>Ie</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>Live</td>
<td>Live</td>
<td>Nightlies</td>
<td>v11 Prev.</td>
<td>Plugin</td>
</tr>
</tbody>
</table>
Media Fragments

• Media-format independent, standard means of addressing media fragments on the Web using URIs
• Allows addressing
  • Time in audio/video
  • 2D Spatial regions
  • Different content channels
• Two formats
  • URI fragments: media.mkv#track=1
  • URI queries: media.mkv?track=1
Multiresolution tile lay-out
Virtual texturing
Rendering with a virtual texture

the tile information is read from the translation table

this determines the location where the tile can be found in memory

the found tile is sampled to get the final result

Tiles stored in memory
Overview

• Introduction
  • WebGL
  • Media Fragments
  • Virtual Texturing
• System Overview
  • Determining the working set
  • Page table generation
  • Page requests
  • Server Side
• Results
• Conclusions
Architecture Overview

Server

HTTP Server Application

Static file hosting
(serves HTML, JavaScript, CSS,...)

Image Server
(serves JPEG tiles, queried using Media fragment URL’s)

Client

Browser

Client Side Visualization
(implemented in JavaScript)

WebGL API
(Part of HTML5 browser)

Graphics Driver (DirectX or OpenGL)

GPU Hardware

HTTP Protocol
Determining the working set

• We have a tiled multi-resolution image but which tiles do we need to download

• Traditional 2D systems (e.g. Google maps, Deepzoom, ...)
  • User set zoom level
  • Current scroll position

• This becomes a lot more complex in 3D
  • Geometry & texture coordinate dependent
  • Occlusion
  • Camera position & orientation
  • Camera field of view
Determining the working set in 3D

- Needs to determine what portions of the texture are needed for every pixel
Page resolver optimizations

• Render the view to a low resolution off-screen buffer
• Low res does not contain information for every pixel
• However
  • Usually highly consistent between neighboring pixels
  • Cache works over multiple frames so it may get requested a few frames later anyway
• Read back this buffer to the CPU
  • Analyze this buffer using JavaScript code
  • Low resolution & type arrays make it fast
Accelerating the translation table

- The list of pages in the cache is converted to a vertex array
- The WebGL is used to rasterize quads into the translation table.

JavaScript code generates quads covering the page in virtual texture space

Results rendered to page translation table

Mipmap-level ordered list of pages
Requesting Tiles

• We request quite a lot of tiles per second
• However we do not need to touch any pixel data using JavaScript
• JavaScript is just a conductor for things going on native code:

```javascript
img = new Image()
... 
img.src = url
```

1. Resolver buffer analyser
2. Send Http Request, Wait for Image Transfer
3. Pass the image to WebGL
4. Upload the Image to GPU Memory, pixel format conversions, ...

```javascript
img.onload = function() {
  gl.texSubImage2D(img, ...)
}
```
Server side

• On the world wide web
  • Custom Apache Module (C++)
  • Less overhead (only a single file handle per server thread)
• In-house
  • Lightweight HTTP frontend interfaces with backend system
  • Dynamically transcodes to JPEG tiles
Media Fragment URI’s

• Use query URIs to retrieve the texture tiles
  • 2D rectangle queries for tiles within a resolution level
  • Track queries for the different resolution levels
  • http://.../example.jpages?xywh=476,596,128,128&track=1

• Advantages
  • Easy to switch between server implementations
  • Pixel based (vs. tile based) addressing hides implementation details
Overview

- Introduction
  - WebGL
  - Media Fragments
  - Virtual Texturing
- System Overview
  - Determining the working set
  - Page table generation
  - Page requests
  - Server Side
- Results
- Conclusions
### Results

<table>
<thead>
<tr>
<th>System</th>
<th>Chrome</th>
<th>Firefox</th>
<th>Opera</th>
</tr>
</thead>
<tbody>
<tr>
<td>Render off-screen buffer</td>
<td>0.03</td>
<td>0.17</td>
<td>0.07</td>
</tr>
<tr>
<td>Read back buffer</td>
<td>10</td>
<td>2.9</td>
<td>6.5</td>
</tr>
<tr>
<td>Analyze buffer</td>
<td>1.4</td>
<td>2</td>
<td>1.2</td>
</tr>
<tr>
<td>Render main view</td>
<td>0.04</td>
<td>0.13</td>
<td>0.21</td>
</tr>
<tr>
<td>Total per frame</td>
<td>11.47</td>
<td>5.2</td>
<td>7.98</td>
</tr>
<tr>
<td>Single tile request</td>
<td>168</td>
<td>422</td>
<td>260</td>
</tr>
<tr>
<td>Single tile load</td>
<td>2</td>
<td>1.3</td>
<td>1.5</td>
</tr>
<tr>
<td>Total per tile</td>
<td>170</td>
<td>423.3</td>
<td>261.5</td>
</tr>
</tbody>
</table>
Results

• Chrome’s out of process architecture suffers when reading back data to javascript.
• It is important to not only look at absolute numbers but also responsiveness
  • E.g. Firefox blocks the user interaction while requesting a page
• Mobile GPUs have to limited precision for our needs
Overview

• Introduction
  • WebGL
  • Media Fragments
  • Virtual Texturing
• System Overview
  • Determining the working set
  • Page table generation
  • Page requests
  • Server Side
• Results
• Conclusions
Conclusions

- WebGL allows us to do very data intensive tasks previously only available on high-end desktop apps
- However there are some hurdles to get it “production ready”
  - Browser performance: 422 ms to request a tile!? 
  - CORS support for <img> resources
  - Asynchronous readback for WebGL (we have had this in desktop API’s for quite a while)
Demo

Visit the demo at:
http://schumann.elis.ugent.be/

Twitter:
@MMLab_Ugent  @cholleme
Bonus Material
Server side scaling

• Many data-intensive sites use separate domains/servers
  • Different software optimized for dynamic/static data
  • Cookie-less domains to optimize http requests
• E.g. Google maps
  • Html & js hosted on maps.google.com
  • Images on a server cloud
    • mt0.google.com
    • mt1.google.com
    • khm0.google.com
    • ...

ELIS – Multimedia Lab
Browser security: Same-Origin Policy

• Only allow http request to the same domain as the script came from
  • E.g. XMLHttpRequest (AJAX)
• Not enforced for pre-JavaScript resources
  • <script>
  • <img>
• Traditionally <img> was not a big issue
  • No way to access the content programmatically
  • Just an abstract DOM object
WebGL <img> restrictions

• WebGL can access image pixels
  • Directly via read-back
  • Indirectly via “performance” leaking
    • P.O.C implementations have been demonstrated
• Could leak the image contents to 3rd parties
• To avoid this, WebGL does not allow creating textures from images downloaded from a different domain
CORS

• CORS = Cross-Origin Resource Sharing
• An extension for the HTTP protocol that allows servers to indicate to the client that sharing the resource with other domains is OK
• The spec works with any resource
• However <img> is only supported in Firefox
• Other browsers support only XMLHttpRequest
  • Work in progress...