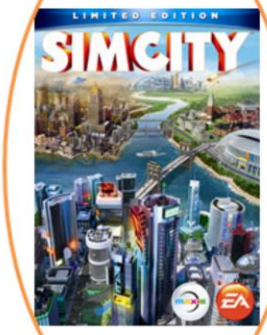
How to Implement AAA Game UI in HTML and JavaScript

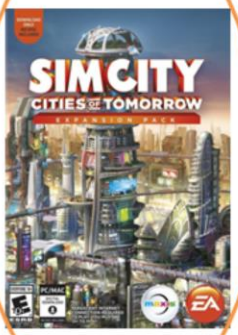
Yee Cheng Chin
Senior Software Engineer, Electronic Arts / Maxis

GAME DEVELOPERS CONFERENCE®
MOSCONE CENTER · SAN FRANCISCO, CA
MARCH 2-6, 2015 · EXPO: MARCH 4-6, 2015

Who am I?

- 7 years at Maxis






CATTO HAVEN POPULATION
Region:
808,483
City:
732,888
Visiting:
2
History
Detail
CITY SIZE
732,888
2
POPULATION SOURCE
Locals
Visitors
POPULATION WEALTH
Low
Medium
High


PLAY

What this talk is about

1. Why HTML*?
2. Maxis' UI, based on HTML. Shipped with SimCity
3. Tips and gotchas

This talk aims at providing the motivations for why using web-based technology (HTML/CSS/JavaScript) is a viable options for implementing in-game UI, and even superior to other technology in some cases. There has been similar talks before in GDC but SimCity is as far as I know the biggest game title to have shipped with purely HTML based UI.

It will then describe and demo MUiLE, our in-house UI system built entirely in web-based technolgy, as an example and inspiration for what is possible and one way to use web-based tech for in-game UI.

1. HTML really means all web-based tech including HTML/CSS/JavaScript and the underlying browser engine
2. EA WebKit + MUiLE, Maxis' UI system. MUiLE is our in house
3. Guidelines, gotchas, general recommendations, random possible optimizations, issues we have encountered during

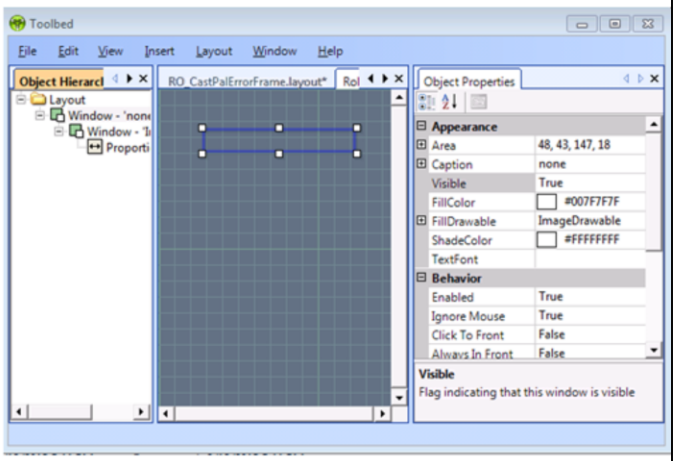
development of our UI system.

What this talk is not about

- Building web games in HTML5. This is specific to using HTML to build just the UI component for a large native game.
- Emscripten / asm.js
- How to build generic web pages.

Maxis' journey in UI tech

- UTFWin
  - Custom built solution, didn't fit all our needs, hard to animate, hook up.



This is just going to briefly go over some points, as this is a talk for why web-based UI is good, not Scaleform vs HTML talk. Ideally throughout the talk the pros and cons would be make obvious.

Maxis' journey in UI tech

- Scaleform / Flash
  - Relies on external license, reliant on Flash.
  - Non-mergeable binary data format
  - Potential issues with ActionScript performance
  - Flash as a general technology was slowly losing support from major players like Apple

SimCity and UI

- Start of SimCity development, needed a new UI system
- Vision of combined web and client interfaces, with shared components between web and in-game UI
- Easy to update and integrate web content

SimCity and UI

- Web-based UI using EA WebKit, and custom-built JavaScript layer.
- Investigated other options as well
  - For engine, at the time didn't find anything better, and EA WebKit was starting to get traction.
  - For building web pages, couldn't find good WYSIWYG editor, built it ourselves.

This video shows clips of SimCity's UI and how they look and work. Includes the front end, loading screen, visualizations (bar graphs/pie charts), and in-game integration to accomplish the "Aero glass" effect for certain translucent panes.

What this talk is about

1. **Why HTML*?**
2. Maxis' UI, based on HTML. Shipped with SimCity
3. Tips and gotchas

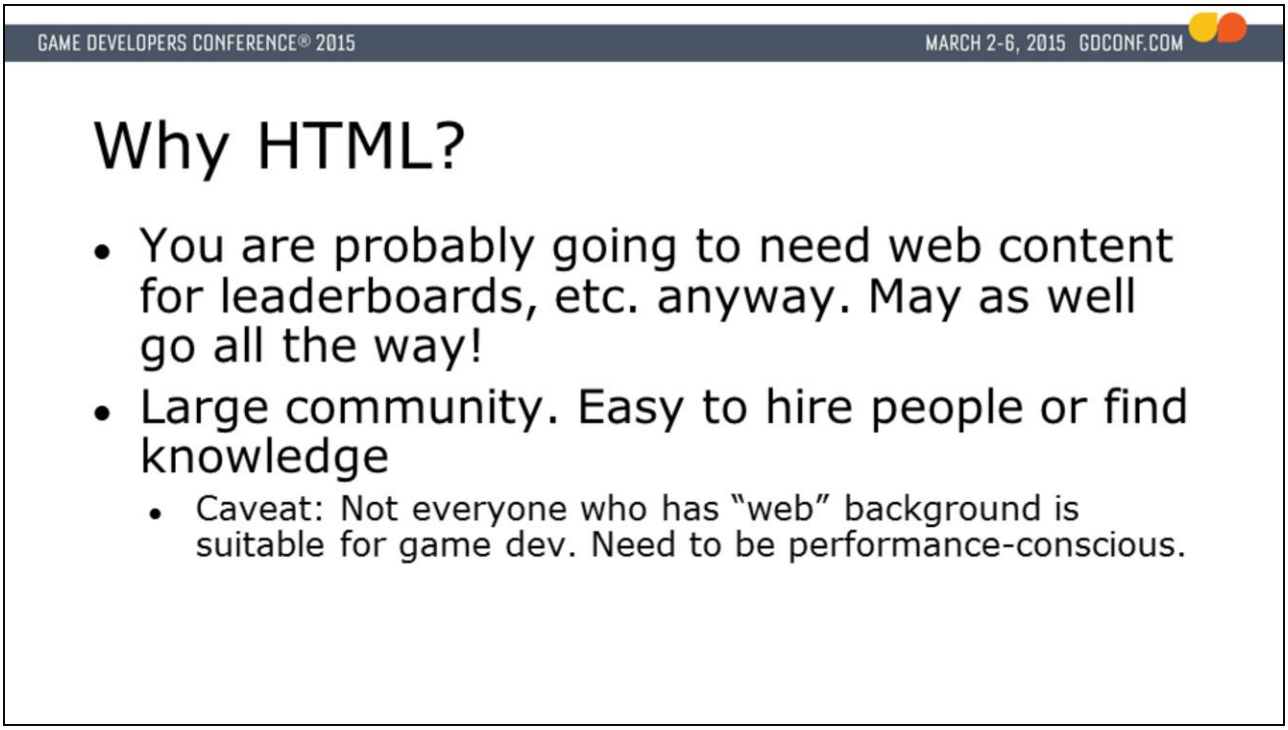
Why HTML?

- Existing tech
  - WebKit, Blink, Gecko, etc.
  - Inspector/Debugger

Among all UI technology, it's fair to say that web-based tech currently has the most development with competing open-sourced implementations coming from Apple (WebKit/Safari), Google (Blink/Chrome), Mozilla (Gecko/Firefox), Microsoft (Trident/IE), etc. The breakneck pace of browser development means it's hard to predict what's going to happen in even just a few years but

Why HTML?

- Fast reloading. Takes 3 seconds to reload the entire UI. No import/export process, or compilation/linking required.
- Most implementations have blazing fast JavaScript engines with JIT compilations.
- Easy to update over the web



One issue with finding advices from the web is that a lot of best practices for web dev aren't designed with 60fps games in mind. A lot of libraries are fully featured but don't translate well to a situation where you need them to terminate within 3ms, so it's essential to establish a standard so people can follow and be conscious that their code needs to run fast.

Why HTML?

- Modding. If you want your PC game to be modded by your community then there's really nothing that beats HTML.
  - Most people know it, and it's easy to modify.
- Caveat: Remember people can and will read your code:
  - `gameCode.DoSomethingStupid= function(stupidity) {… /* ☹ */}`

Mention population display code being discovered by players.

Other EA games using HTML tech

- SimCity
- Skate 3
- Sims
- Most new console games for online features

What this talk is about

1. Why HTML*?
2. **Maxis' UI, based on HTML. Shipped with SimCity**
3. Tips and gotchas

SimCity's UI

- More complicated than action oriented games UI.
  - As a result both have more requirements (layouts, dynamic UI scaling, etc) and higher budget.
- EA WebKit
- MUiLE

EA WebKit

- Backend of our UI
- Fork of Apple's WebKit project, but designed to be embedded into games, while providing much more hooks such as custom memory allocator, profiler, JS/C++ bindings, network layer, etc.
- Open sourced: http://gpl.ea.com

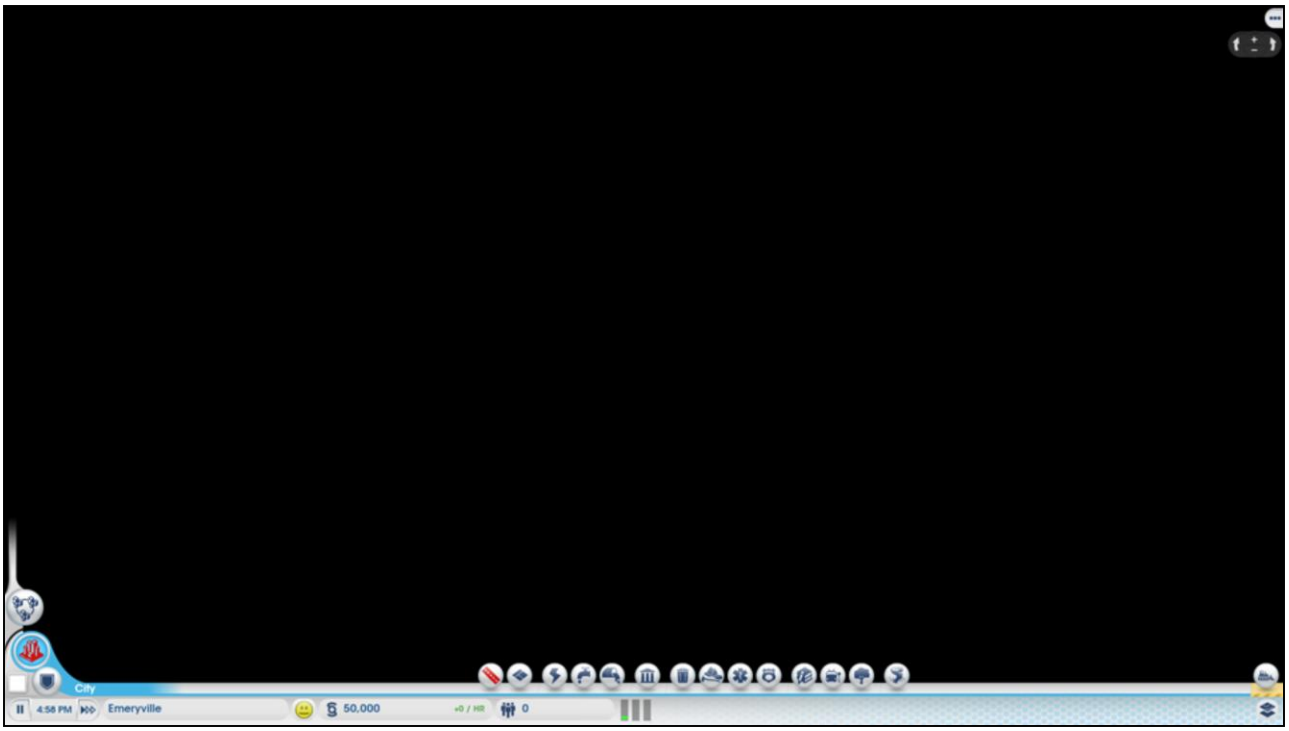
EA WebKit

- Get all the benefits of active development (and drawbacks).
- WebKit's modular design helps adapt to other platforms.
- Good inspector for live inspection, JavaScript debugging
- Gotcha: Doesn't work on mobile (platform limitations)

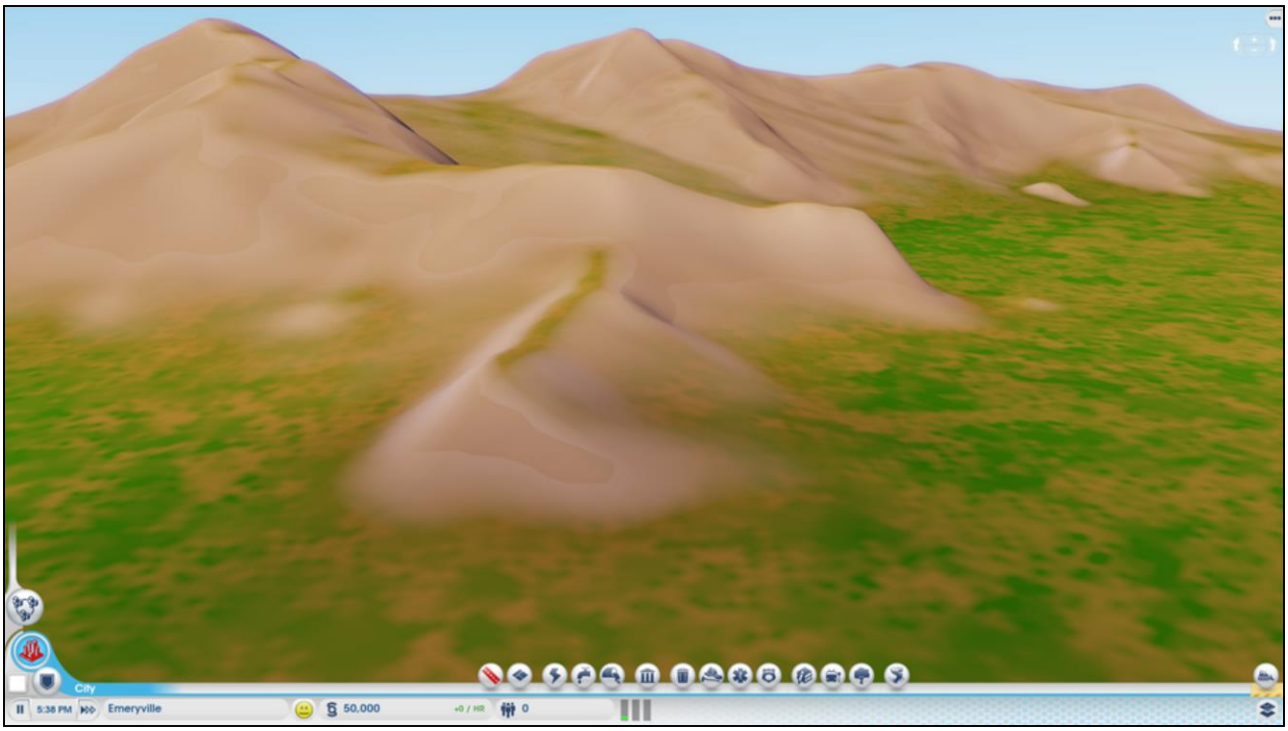
WebKit has shown to be a very successful project, and used in multiple platforms, OS, from phones to desktops. One caveat is that you will ultimately be adding an additional dependency on companies developing it.

EA WebKit

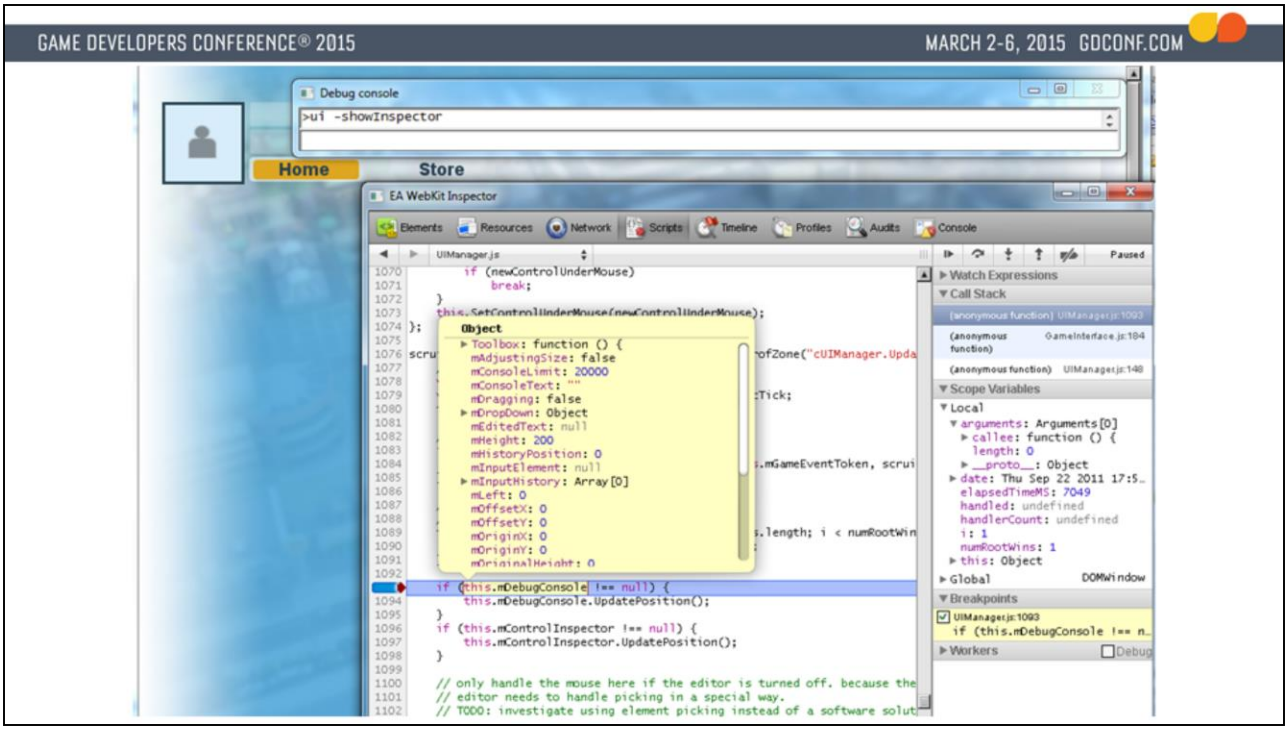
- Features:
  - Supports multiple views
  - Hardware compositing API
  - Efficient JavaScript bindings
  - Designed for games, support plugins for custom text renderer, memory allocators, etc
  - (First party support)



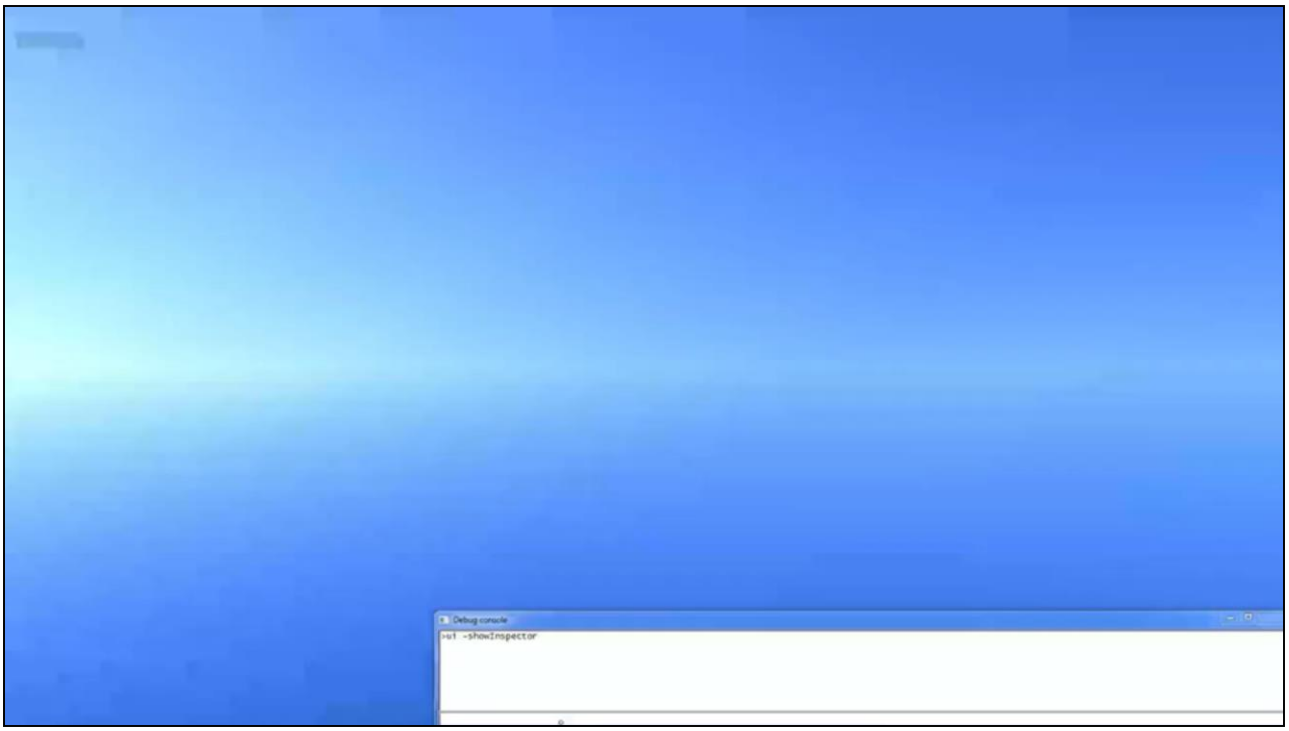
EA WebKit just outputs to a plain 2D texture that we then renders on top of the 3D scene. Background uses “transparent” CSS background style to make the content show through.



Final composite rendering the UI half transparent texture on top of 3D world.



The inspector is virtually identical to the one in Safari, so it's very familiar with web developers.



This video shows

1) how we can easily bring in external web pages (in this case google.com) into our in-game UI
2) The in-game inspector (the same as the one used in Safari) can be used to inspect elements, and also modify dynamic states via the JavaScript console

MUiLE

- HTML/JavaScript-based UI layer.
- Custom to Maxis.
- Built most of the functionality from the ground up as we couldn't find good alternatives at the time.
- Just implementing a button with the correct behaviors took some time…

MUiLE

- All UI 100% in HTML/JS/CSS.
- Component based, storing layouts in JSON files, which allow us to merge and allow concurrent edits.
  - Layout files then loaded in dynamically and the DOM is constructed from them.
- Layouts can link to other layouts, allowing reusability.

```
{
 "instanceID": 1,
 "left": 205,
 "top": 98,
 "width": 800,
 "height": 600,
 "visibility": true,
 "ignoreMouse": true,
 "children": [ {
   "instanceID": 2,
   "left": 10,
   "top": 10,
   "width": 176,
   "height": 45,
   "visibility": true,
   "drawable": {
    "type": 2,
    "images": [ "Graphics/textInputField.png" ]
   },
   "type": "cWindow"
  },
```

```
{
   "layoutPath": "Layouts/GlobalUI2.js",
   "instanceID": 3,
   "left": 0,
   "top": 0,
   "width": 800,
   "height": 600,
   "controlID": 174136993,
   "visibility": true
  } ],
 "type": "cLayout",
 "version": 1
}
```

Example of what one such JSON file looks like. It represents a scene graph with each node possibly having other children, and can also reference other layout JSON files (with a .js extension).

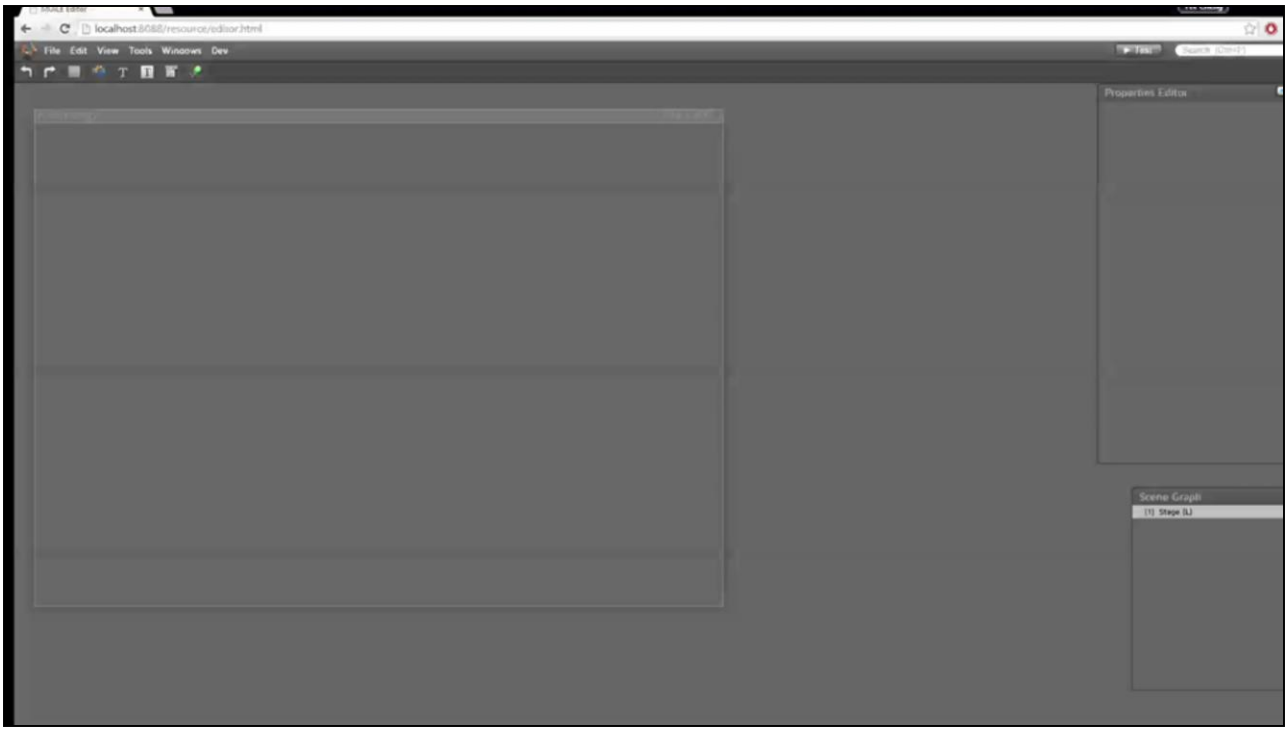
MUiLE

- In game communication is done through async callbacks, through game commands, game events, and game data callbacks.
  - Better for multithreading, and similar to the async nature of web interfaces
  - `PostGameCommand(kCmdDoSomething, someData, function(result) { /* got result! */});`
  - `RequestGameData(kDataPopulationCount, function(data) { /* process data */ });`

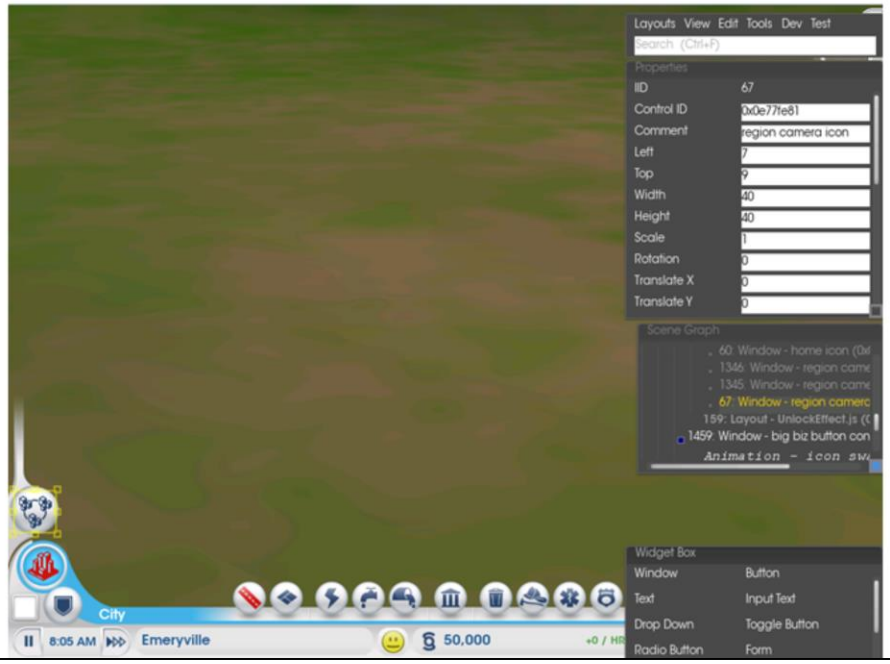
The code is a little simplified from how it works in our codebase but the gist is the same. In fact we didn't eventually multithread our UI code but having them all run through an async API makes it easier to migrate in the future.

MUiLE Editor

- WYSIWYG editor, also built in HTML as part of the package itself, allowing it to be used in any browser.
  - No dependencies. Anyone with a debug version of the game can edit UI using a browser.
  - Can edit the UI in-game.



Editor demo video, showing off the tool we use to build UI. The editor is entirely built as a web page, served from a dev version of the game (which acts as a localhost server).



Screenshot of in-game editor.

MUiLE Editor

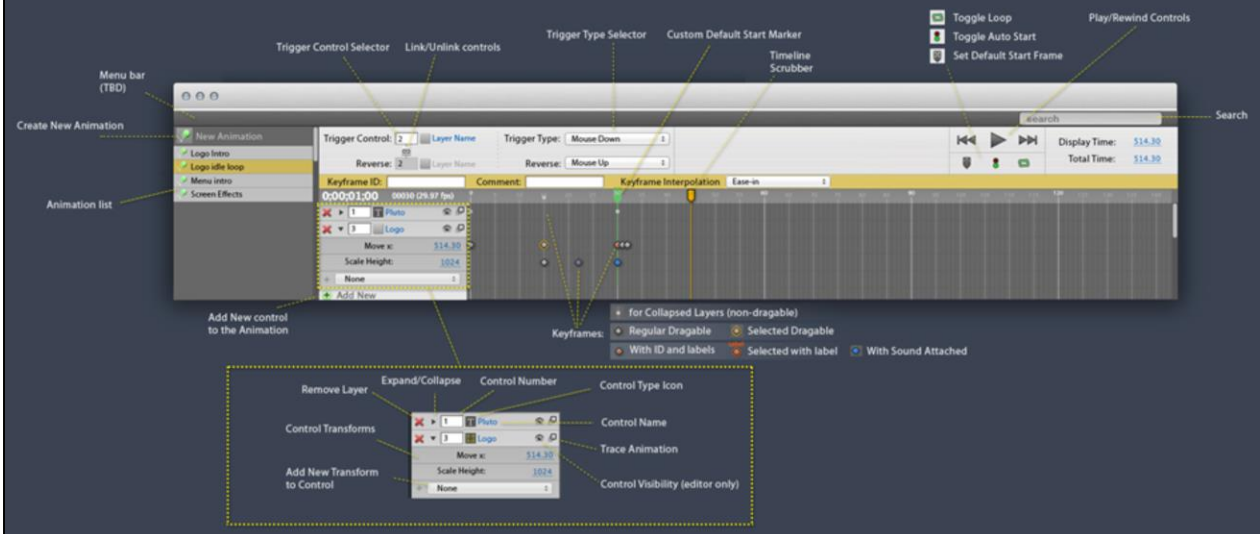
- Communicates with game through a localhost server served by the game (we already use that for other debugging utilities)
  - Uses a REST-like API. When in game we expose a custom URL handler (game://localhost/) instead.
  - Editor: `http://localhost/resource/editor.html`
  - List all the layouts: `http://localhost/dir/layouts` (GET)
  - Save layout: `http://localhost/layout/mainMenu.json` (POST)
  - Load layout: `http://localhost/layout/mainMenu.json` (GET)

Requiring the game running is actually one of the annoying points of using this editor. Standard web pages cannot save and load files from the file system which is why we had to run that through a native app to do that. Another option would be to write a simpler command line app, or use something like Chrome Apps which allow more flexibilities in terms of dealing with file system.

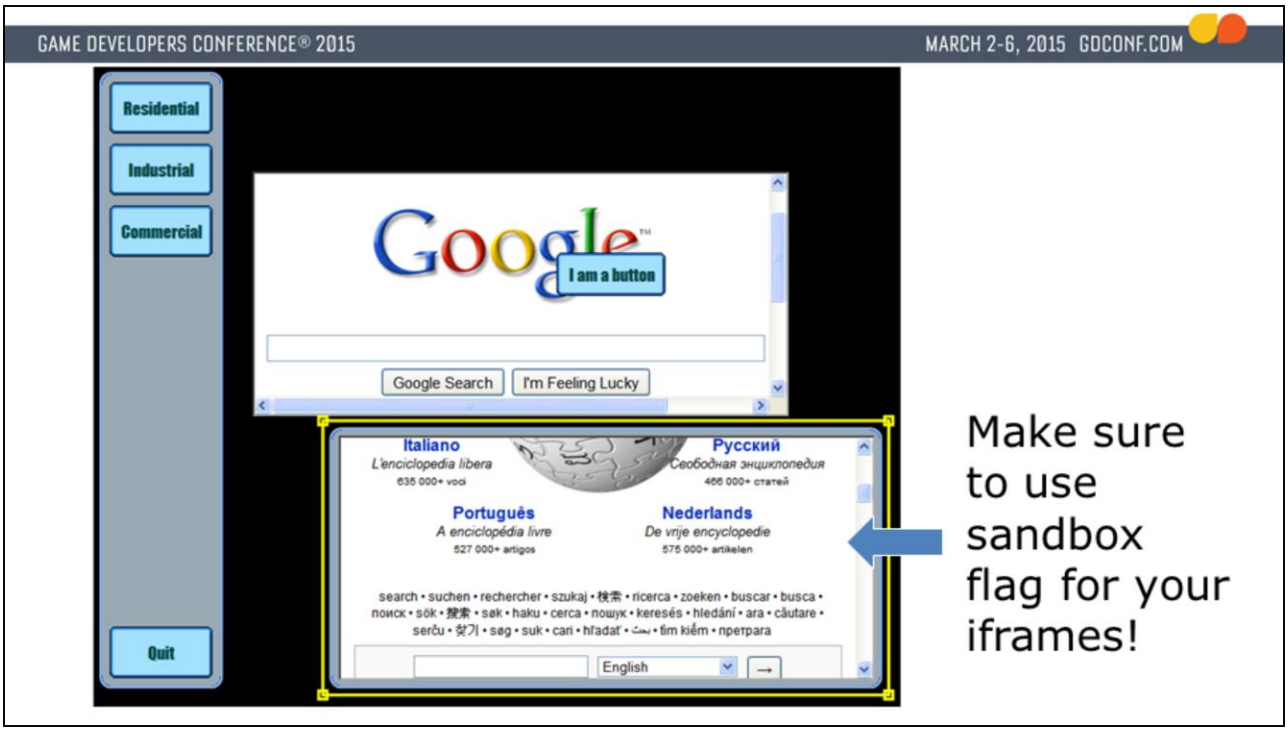
MUiLE Animations

- Recommended way is to use CSS
  - Fade in/out, transitions, keyframes etc.
  - When we started it wasn't as advanced, and we found out we also wanted more control.
- Implemented custom animation system
  - Full control over timeline, can scrub, stop, loop.
  - Controls exactly the parameters we need.
  - Probably increased load in the JS engine as they aren't natively animated like in CSS.
  - Each control has triggers to play/stop/loop animations. Each animation is a timeline of different controls' states such as positions, visibilities, rotations, or game events.

UI Tool Timeline



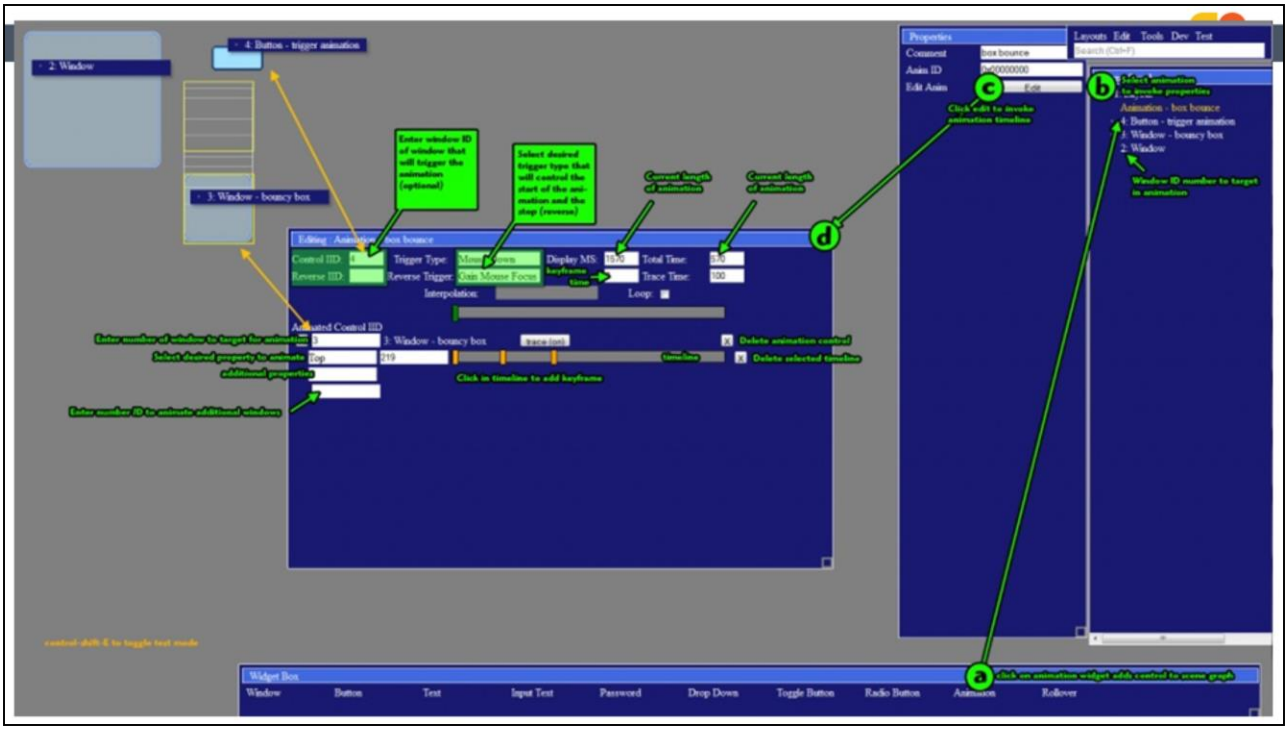
Early Sceenshots



One of the early things we experimented with was to embed live web content, which turned out to be quite easy to do, as it's just one iframe away.

Unfortunately this experiment did not go as far as we hoped and we ended up not embedding external web content in our game UI.

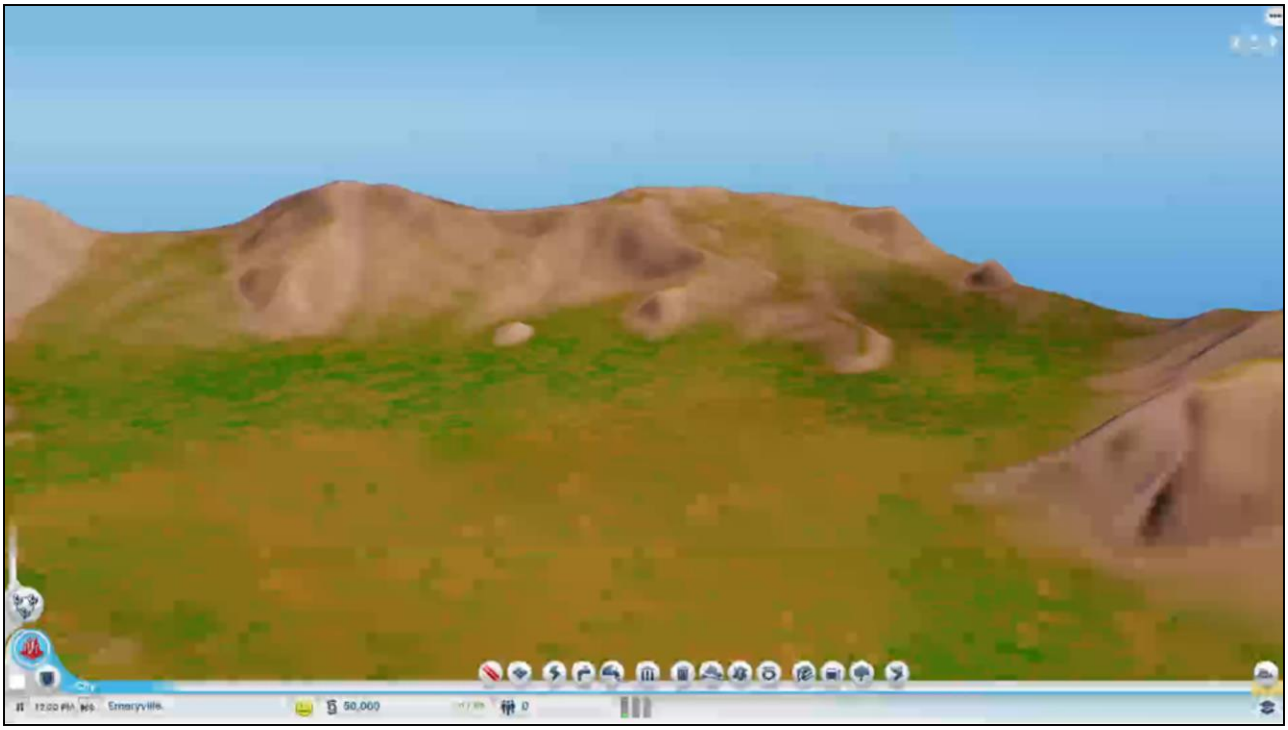
One caveat with doing it was that some external web pages would refuse to be loaded in iframes and took over the entire page for security reasons. Nowadays, this is more controllable sine you can use the sandbox flag on iframes to prevent frame jacking.

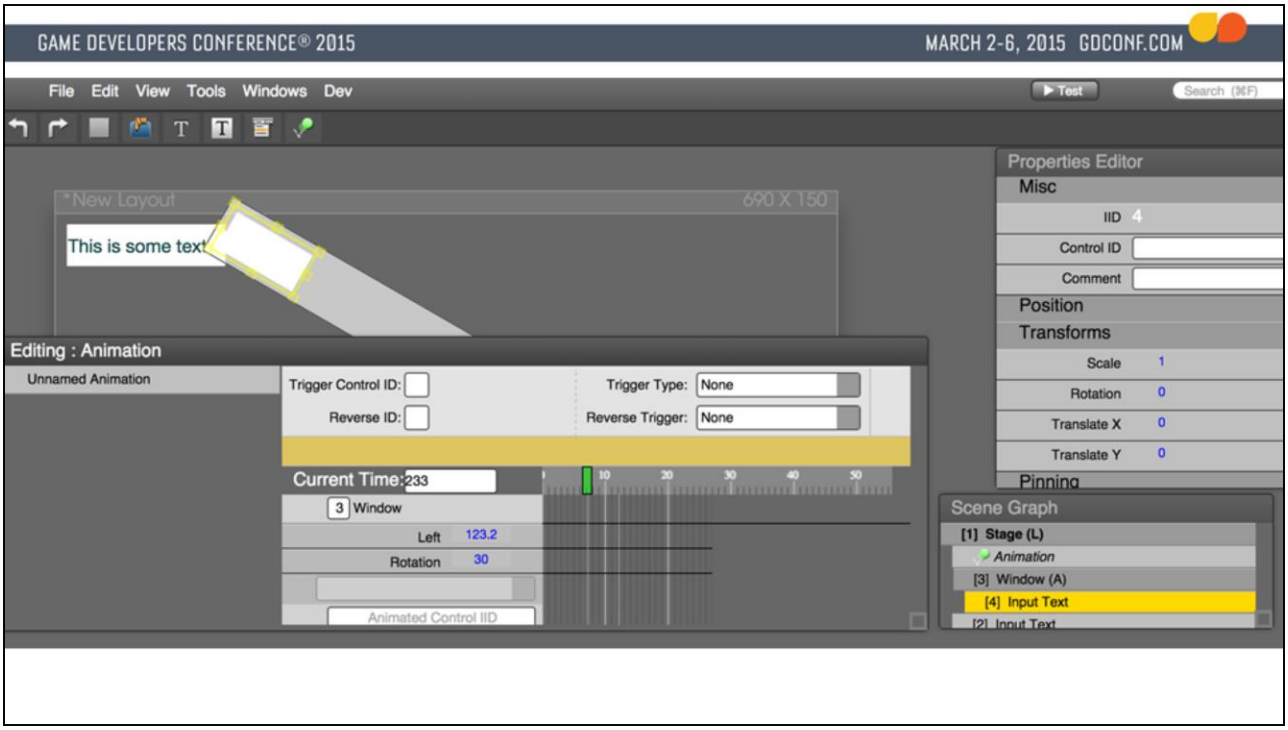


Mid-way documentation of editor. Shown in the middle was the animation editor that was a work in progress.


Select a Text Style
textToolName
Something about a lazy dog.
textToolCost
Something about a lazy dog.
textWhiteCaptionCenter
Something about a lazy dog.
textWhiteCaptionLeft
Something about a lazy dog.
textButtonLabel
Something about a lazy dog.
textButtonLabelShadow
Something about a lazy dog.
textButtonFilterLabel
Something about a lazy dog.
textGraphLabel
Something about a lazy dog.

Another experiment was in world UI. What the screenshots showed were semi-transparent web pages being drawn in world space. This turned out to only be OK in certain limited sense, and while it was cool texts turned out to be much more legible if it's just plain 2D UI aligned correctly without aliasing issues.


Emeryville
50,000
0



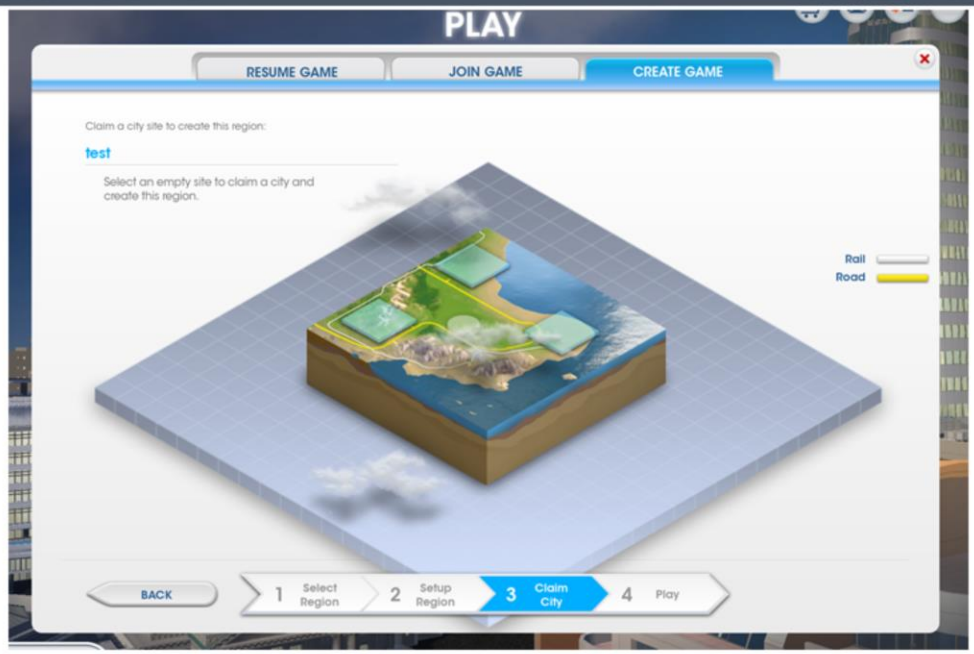
Current version of editor

Video showing hotloading layouts that take couple seconds and don’t require a repack step at all.

Final Shipped UI

Draggable launcher window.

There are actually two web pages here. The main UI is one, and the billboard saying "Offline is here" is actually another web page rendered as in-world UI.



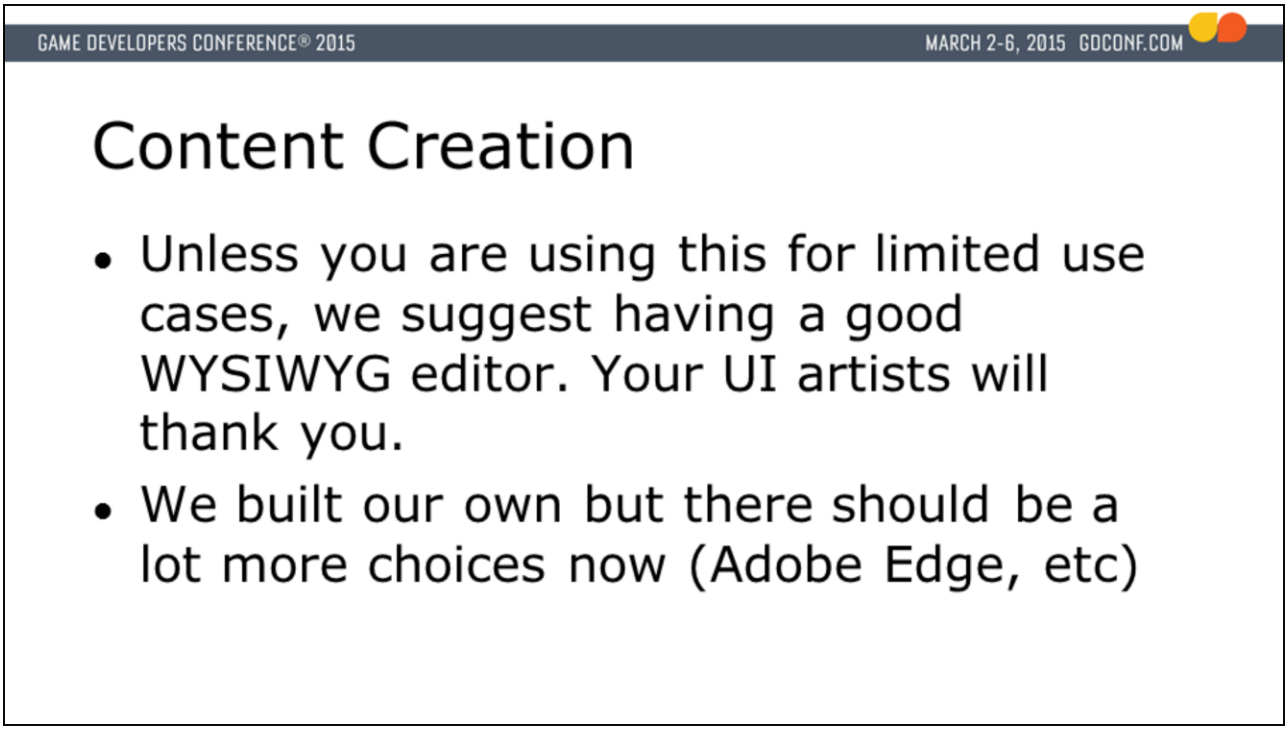
This is the page with the scrolling clouds shown earlier.


Breaking Ground
Draw a road from the highway
Zone residential and have 10 more houses
Roads
Guides
City
12:16 PM
Trinity Point
1,000,000
+0 / HR
0

Chart drawing using third party libraries.

What this talk is about

1. Why HTML*?
2. Maxis' UI, based on HTML. Shipped with SimCity
3. **Tips and gotchas**



One issue with web-based UI has always been that it's very programming heavy. HTML/CSS/JS are all text-based formats and past WYSIWYG tools haven't done a very good job in allowing artists to make dynamic web pages, and nowhere as easy as Flash. This was the first thing we identified that we needed to solve in order to create UI as good as other competitive games.

It's also very hard to beat Flash in this area since ease of creation is the strength of that ecosystem. If you have artists coming from using Flash the best you can is to ease their pains and use the best tools you can find.

Adobe has also been building tools like Adobe Edge to break into HTML5 animations, so the market is getting more mature. There are also other smaller tools such as Tumult Hype with more limited functionality.

Content Creation

- Traditional split of HTML/CSS/JS is such that they represent content, style, and logic. Good for representing documents.
- Essentially building a mini-application. Built most UI out of JSON modules that are loaded in dynamically through JS instead.

We decided HTML is primarily designed as a document format, it's best not to get too hung up on its semantics when you are building an application. As a result we rely heavily on JavaScript to dynamically generate UI, and link to CSS for styling.

Pick a Good Engine

- Access to source code. You really need to be able to dig into it if things go wrong.
- Find one which you can get good support from.
- Allow custom memory allocator hooks, etc. You need the control.
- Has hardware rendering support and provides hooks for it.
- Comes with a standalone demo app for you to test pages on.
- Supports JIT compilation
  - JIT mode is at least twice as fast for us

Having source access allowed us to identify memory allocation issues, identify inefficiencies in C++/JS bindings, etc, which would all not be possible if it was a black box.

There are a few options for embedding web content now. There is Chromium Embedded Frameworks (https://code.google.com/p/chromiumembedded/), EA WebKit's open sourced portions, etc.

JavaScript Organization

- Be careful with common libraries such as jQuery.
  - They may be great for web development with tons of features, they may not give the best per-frame performance or memory-use.
  - Make sure to profile before you commit!
- Read *JavaScript: The Good Parts*

Profile, profile, profile, profile, profile!

JavaScript Organization

- We used the Google Closure Library
  - Library developed in a modular fashion, allowing you to selectively pull in only the necessary components.
  - Contains useful functions for matrix calculations, cryptography, basic utilities for inheritance, etc.
  - Open Sourced

One flaw with JS is that it doesn't really come with a standard library. The system-provided APIs tend to be limited and it's up to the developer to integrate other third party libs. We have found that Closure provides a large set of functionality that covers a wide range of features and the compiler then allows us to cherry pick the feature we need instead of linking every piece of code in.

JavaScript Organization

- Google Closure Library (continued)
  - Solves the issue of managing large amount of JS files. Other solutions usually involve just concatenating them all or modifying HTML files.
  - Allows you to specify dependencies among JS files and build up a manifest JS files that pull in all necessary dependencies.
    - May be less useful with advent of Common JS.

With the advent of Common JS, the way Closure specifies dependencies and pull in other JS files may be outdated. If you are starting a new project now it would be wise to look into that first.

JavaScript Organization

- Standard JS annoyances

```
<html><body>
    <script src="ControlInspector.js"></script>
    <script src="UIAnimationEdito.js"></script>
    <script src="UIEditorDropDown.js"></script>
    <script src="UIEditor.js"></script>
…
```

Standard way of managing JS files are messy, hard to reuse across html files, and difficult to manage dependencies and order of inclusion

JavaScript Organization

- Google Closure Library dependencies example

Project.js:

```
goog.provide('muile.editor.project');

goog.require('muile.project'); // pull in the general muile library

goog.require('muile.editor.ControlInspector');
goog.require('muile.editor.UIAnimationEditor');
goog.require('muile.editor.UIEditor');
goog.require('muile.editor.UIEditorDropDown');
goog.require('muile.editor.UIEditorProperties');
...
```

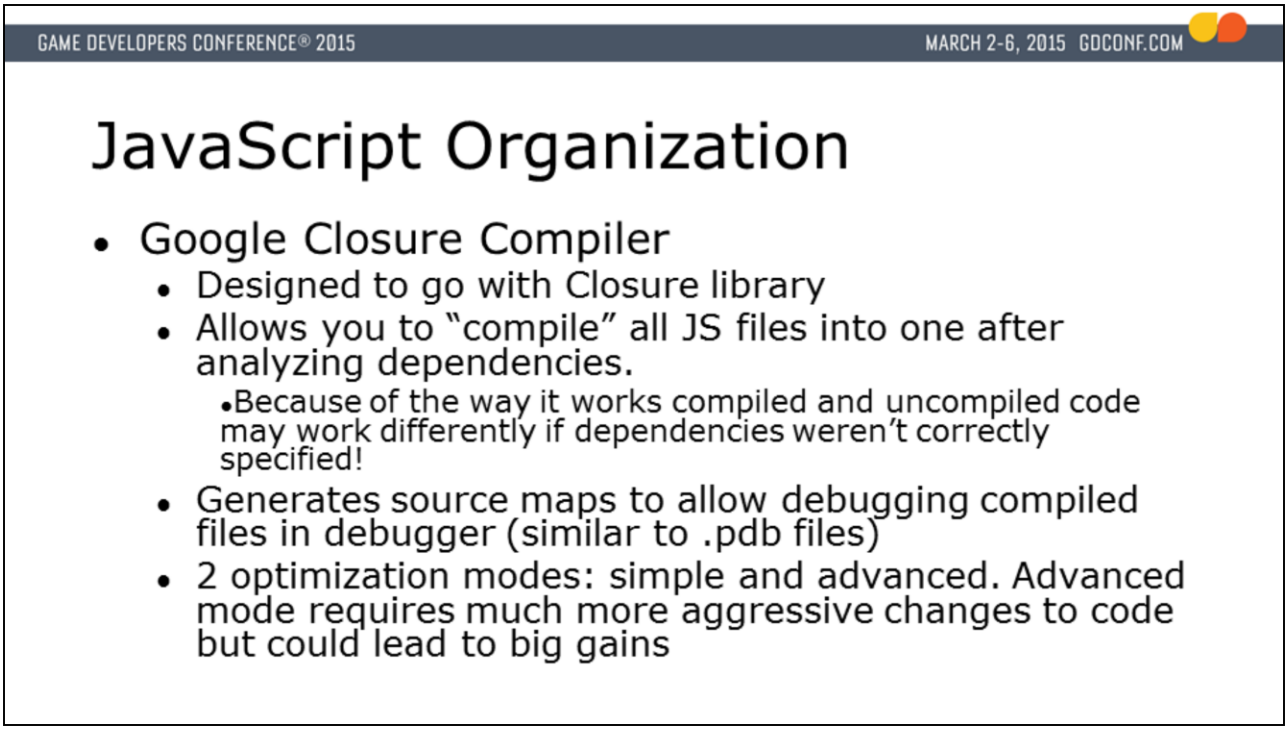
EditorControlInspector.js:

```
goog.provide('muile.editor.ControlInspector');
...
```

Editor.html:

```
<script src="Project.js"></script> <!- this automatically pulls in the other files -->
```

Example how the dependencies were specified. Here is the manifest for the editor code, which pulled in the main MUiLE components, and then pull in all the editor related code. The main MUiLE components dependencies are defined in another file that specifies "goog.provide('muile.project');"



The switch to Advanced mode deserves a little more discussion. For a lot of JS codebase it would involve a fairly intrusive refactor that may go against some established JS coding styles. The idea is to allow the compiler to perform as much inlining and optimizations as possible and reduce dynamic reflection, but a lot of common idioms such as intermixing myObj.myMember and myObj[`myMember'] would now cause bugs because myObj.myMember would get optimized statically. For more info see https://developers.google.com/closure/compiler/docs/api-tutorial3

As for whether it's worth it or not we have found the result to be mixed and eventually didn't pursue it, partially because of the push back from the team due to the amount of work we had to put in. In one test we did that was very JavaScript and function call heavy (spawning hundreds of animations, all calculated in JS) we could see up to 30%-40% faster performance. However in other more common cases that mostly spent time dealing with the DOM etc it was not as significant. The other main benefit is the size reduction but

given the amount of memory modern machine has and we're storing these JS files locally it's not as big an issue for us.

Whether a team would want to switch over or not would probably then depends on the existing codebase and preferred coding style, and also how heavy they are doing work in JS.

Another issue was that compiled code (even Simple mode) tends to be annoying to debug. We have source maps to help but while it provides a mapping to the original source location it's hard to inspect or modify the code dynamically like in raw JS. Incorrectly dependencies could mean you have different behaviors in compiled vs non-compiled code as well.

JavaScript Organization

- Google Closure Compiler (continued)
  - Examples of advanced mode compilation:

```
var DEBUG = false;
var counter = 1;
if (DEBUG) {
  console.log('Super secret output:' + counter++);
}
console.log('Generic boring output:' + counter);
```

  - *Compiled to:*

```
console.log("Generic  boring  output:1");
```

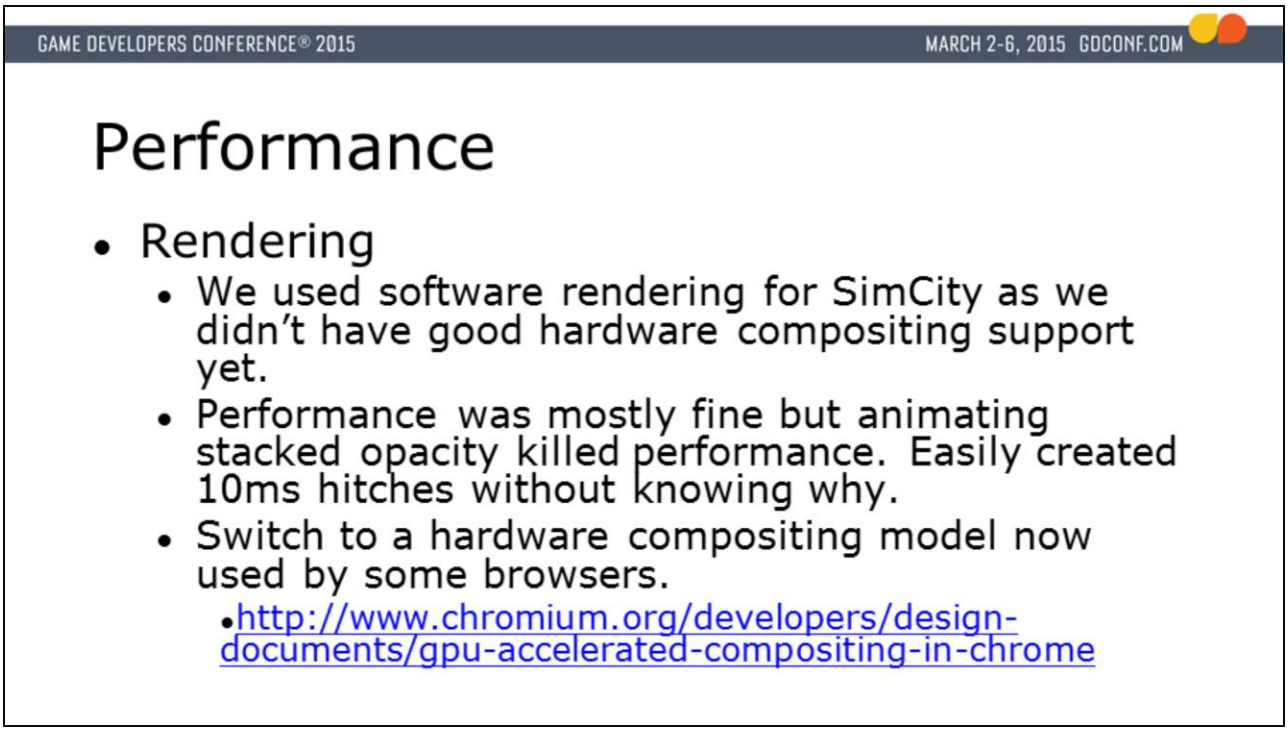
Dead code removal could be useful for removing debug scripts from accidentally leaking.

It's possible to use other macro systems to accomplish the same thing but this allows us to use JS syntax and doesn't rely on an external macro language.

JavaScript Organization

- Communicating with the game
  - Use C++ bindings
  - You could try to be cute and use REST APIs
    - `game://localhost/Game/Commands/Sim/AddPopulation/1`
  - You are kind of adding unnecessary cost for string parsing etc. Just call the C++ function.
    - `Game.AddPopulation(1)`

The “Game” object here is a C++ runtime object bound to JS.



Software rendering means WebKit just renders everything to a texture for you and you just present it. It's much more simple to integrate but does limit what you can do with it, as the GPU tend to be more more efficient at doing things like blending and transforming textures.

Performance / Rendering

- Make sure can do dirty rect visualization.
- If you are using WebKit or Blink based browsers, use translateZ(0) to force elements into another layer if using hardware compositing. Only do this for animating elements.

In the screenshot, the button pointed by the arrow would blink and radiates a red circle around it. The animation will repeat continuously until it was dismissed. This innocuous-looking animation was hitching the game and took us to a while to identify the trouble spot. The constant triggering of the animation plus the fact that it was blending over other opacity areas meant the software rasterizer had to redraw a lot of content. With a hardware compositor this should be much more easily solvable.

We used to have another example which was worse. When the game was paused we would draw a ring around the whole screen. Since that ring would be blinking, every frame it had to re-blend with the whole UI causing everything to be redraw, costing at least 10ms per frame.


City
7:47 AM
Emeryville
50,000
+0 / HR
0

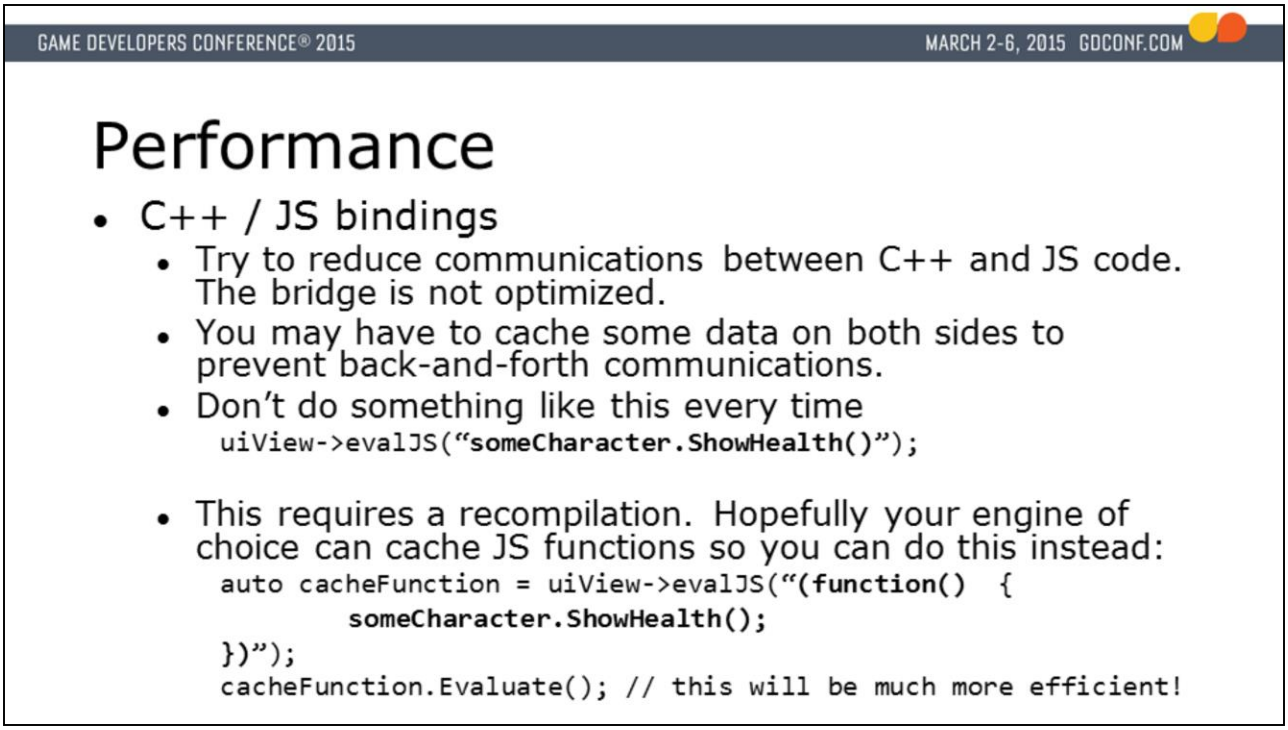

City
8:20 AM
Emeryville
50,000
+0 / HR
0

Performance

- Memory use
  - Watch your memory use! We found that it's not easy to profile memory used by the UI system as we would get a global heap using up to 100mb of memory, with no finer details. Blink-based browsers seem to have better control over this.
  - Try to use pools instead of dynamic allocation as much as possible

Profile profile profile! This is one of those cases having source access is useful. You don't really want to do it but if you are trying to optimize the last bits of memory it's useful to step through the code in the browser to know what it's doing.

Also access to source code would allow you to accurately predict how much memory alloc happens when you call C++ <-> JS, etc. Also what kind of size of block they allocate so you can use a fixed allocator for them.

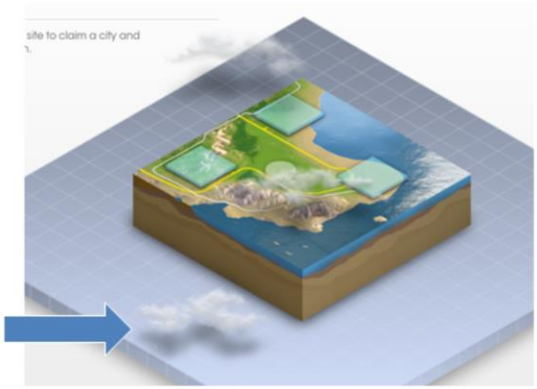


If you think about how JS compilations work it would be easy to see why calling into C++ is expensive even if it's implemented efficiently. Since you are crossing the language boundary it's hard for the JS runtime to reason about the side effects of your C++ code, and thus it can't do a lot of aggressive optimizations on the JS side around the C++ binding call. It's for this reason that Google's new Blink project is experimenting with moving more of the DOM to be processed in JavaScript.

Compilation of JS dynamically is expensive. The second piece of code caches a function pointer and usually the engine will keep the compiled results every time you evaluate it. The results are usually JIT compiled so calling into JS functions this way tend to be a lot more efficient.

Grab bag

- Scrolling text and images smoothly is surprisingly hard!



Different browsers handle CSS scrolling differently. Some browsers interpret fractional CSS positions correctly and do the proper antialiasing, while others would round it. Instead of using CSS positions you could also use CSS transforms (using translation). Depending on the browser, this rounding behavior may or may not be the same between CSS position and CSS translation.

The demo will first show web page that uses both kinds of translations and also canvas in different browsers and how they behave differently. After that show the in-game scrolling to get some contexts of the scrolling clouds.

This is really an example that different web browsers often have completely different set of best practices that are contradictory with each other. You will need to decide which ones you want to support and sometimes come up with browser specific solutions, unfortunately.

Non-PC platform issues

- Console
  - Mostly works, but you won't get JIT-compiled JavaScript code. Reduce JS workload and budget accordingly
- Mobile
  - Especially on iOS it's not possible to ship your own HTML/JS runtime, so need to use native web view.
  - iOS now supports JIT through WKWebView

Wrap up

- It's possible to make quality UI using HTML
- Tools and libraries available.
- Building our own tools and editor was very time consuming.
- Performance *will* be less than native UI
- Improved iteration time and ease of development was worth it.

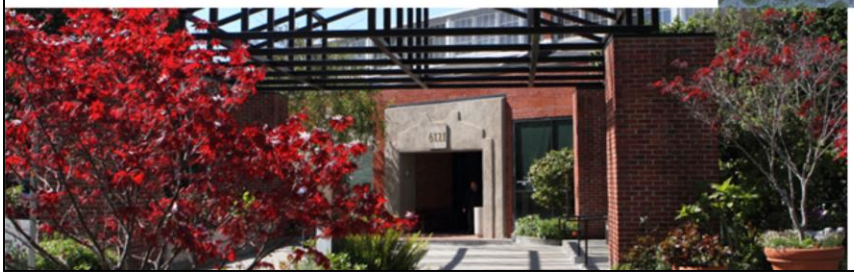
Since we didn't have hardware compositing support and also had to write a new editor, turned out we couldn't do everything we wanted. We think the end results were worth it but it is a real cost to consider.

Performance included: rendering perf, memory, general JS

Thanks!

- **Brad Smith** and **Scott Clarke**, who did a lot of the actual work on this.
- **Renaud Ternynck** for his continuous bombardment of request for features and improvements.
- **EA WebKit team** for their support throughout.
- The entire **Maxis UI team** for making this all possible.

R.I.P...



ychin@maxis.com

Q & A