Acta Cybernetica **21** (2014) 291–306.

Lively3D: Building a 3D Desktop Environment as a Single Page Application

Jari-Pekka Voutilainen*, Anna-Liisa Mattila*, and Tommi Mikkonen*

Abstract

The Web has rapidly evolved from a simple document browsing and distribution environment into a rich software platform, where desktop-style applications are treated as first class citizens. Despite the associated technical complexities and limitations, it is not unusual to find complex applications that build on the web as their only platform, with no traditional installable application for the desktop environment – such systems are simply accessed via a web page that is downloaded inside the browser and once loading is completed, the application will begin its execution immediately. With the recent standardization efforts, including HTML5 and WebGL in particular, compelling, visually rich applications are increasingly supported by the the browsers. In this paper, we demonstrate the new facilities of the browser as a visualization tool, going beyond what is expected of traditional web applications. In particular, we demonstrate that with mashup technologies, which enable combining already existing content from various sites into an integrated experience, the new graphics facilities unleashes unforeseen potential for visualizations.

Keywords: web apps, visualization, window management, 3D UI

1 Introduction

Over the few recent years, the Web has evolved from a simple document browsing and distribution environment into a rich software platform, which is capable of hosting desktop-style applications. Moreover, these applications are increasingly often treated as first class citizens.

The document-centric origins of the Web are still visible in many areas. Consequently, it has been traditionally considered difficult to compose truly interactive web applications. A partial solution has been to use plug-in components or browser extensions, such as Adobe Flash or Microsoft Silverlight, but such binary or company specific technologies do not fit well to the ideals of the open web, advocating web applications that are built using technologies that are open, accessible and as

*Department of Pervasive Computing, Tampere University of Technology, E-mail: {jari-pekka.voutilainen, anna-liisa.mattila, tommi.mikkonen}@tut.fi

DOI: 10.14232/actacyb.21.3.2014.2

interoperable as possible to avoid vendor-specific lock-in. As a manifestation of this attitude, it is not unusual for complex applications to use the web as their only platform. In other words, despite the technical difficulties and limitations, there is no traditional installable application for the desktop – the system is simply accessed via a web page that is downloaded inside the browser, whose runtime resources are then used by the application. We believe that the transition of applications from the desktop computer to the web has only started, and the variety, number, and importance of web applications will be constantly rising during the next several years to come.

In comparison to desktop applications, the benefits of web applications are many. Web applications are easy to adopt, because they need neither installation nor updating – one simply enters the URL into the browser and the latest version is always run. Furthermore, web applications are easy and cheap to publish and maintain; there is no need for intermediates like shops or distributors. Furthermore, in comparison to conventional desktop applications, web applications have a whole new set of features available, like online collaboration, user created content, shared data, and distributed workspace. Finally, with the whole content of the web acting as the data repository, the new application development opportunities, unleashed by the newly introduced facilities of the web technologies that make the browser increasingly capable platform for running interactive applications, are increasing the potential of the web as an application platform.

In this paper, we demonstrate the new facilities of the web as an information visualization tool, going beyond what is expected of browser based applications. Moreover, we demonstrate that together with mashup technologies, which enable combining already existing content from various sites into an integrated, usually more compelling experience, the new graphics facilities results in unforeseen potential for visualization of context-specific data. Together with data science, the approach can be generalized to increasingly complex systems, which simplifies data consumption tremendously.

The rest of the paper is structured as follows. In Section 2, we discuss the evolution of the web and the main phases that can be identified in the process, and briefly address two important web standards - HTML5 and WebGL - and their role in the development of new types of web applications, building on already available resources. In Section 3, we introduce our technical contribution, Lively3D, which is a host environment that is capable of integrating multiple applications within single 3D-scene and visualize the environment in three different ways. In Section 4, we discuss development issues related to Lively3D's 3D user interface and introduce a redesigned version of Lively3D's UI. In Section 5 final conclusions are drawn.

2 Background

The World Wide Web has undergone a number of evolutionary phases [6]. In the first phase, web pages were truly pages, and navigation between pages was based simply on hyperlinks – a new web page was loaded from the web server each time

the user clicked on a link. These pages were truly page-structured documents that contained primarily text with some interspersed static images, without animation or any interactive content, which were only introduced in the second phase, as web pages became increasingly interactive, created by using animated graphics and plug-in components. In this phase, the JavaScript scripting language enabled building animated, interactive content with technologies primarily associated with the Web only. Moreover, as a part of the transition to this phase, the Web started moving in directions that were unforeseen by its designers. Web sites started behaving more like multimedia presentations rather than page-structured documents, content mashups and web site cross-linking became increasingly popular.

Today, the browser is increasingly used as a platform for real applications, with services such as Google Docs with its desktop-like interactions paving the way towards more complex systems. We expect that as more and more data becomes available online, the capabilities of the browser will be increasingly often harnessed to filter and further process the data into a form that can be more easily consumed. In this context, two recent initiatives form an important perspective. These are the open web, perhaps best manifested in Mozilla Manifesto[1], which centers around the idea that the web that is a global public resource that must remain open, accessible, interoperable and secure, and open data, which according to Wikipedia[2], builds on the idea that certain data should be freely available to everyone to use and republish as they wish, without restrictions from copyrights, patents, or other mechanisms of control.

To support the above initiatives, the need to use plugins is being seriously challenged by two recently introduced technologies, HTML5 and WebGL, as already pointed out in [5]. These new technologies provide support for creating desktop-like applications that run inside the browser (HTML5) and enable direct access to graphics facilities from web pages (WebGL). This, together with already well-known techniques for mashupping, are paving the way towards the next generation of web applications, with increasing capabilities for modeling and visualizing data and conceptual information.

The forthcoming HTML5 standard[3] complements the capabilities of the existing HTML standard with numerous new features. Although HTML5 is a general-purpose web standard, many of the new features are aimed squarely at making the Web a better place for desktop-style web applications. There are numerous additions when compared to the earlier versions of the HTML specification. To begin with, the new standard will extend the set of available markup tags with important new elements. These new elements make it possible, e.g., to embed audio and video directly into web pages. This will eliminate the need to use plug-in components such as Flash for such types of media. The HTML5 standard will also introduce various new interfaces and APIs that will be available for JavaScript applications.

[1] http://www.mozilla.org/about/manifesto.html
[2] http://en.wikipedia.org/wiki/Open_data
[3] http://www.w3.org/TR/html5/

WebGL[4] is a cross-platform web standard for hardware accelerated 3D graphics API developed by Mozilla, Khronos Group, and a consortium of additional companies including Apple, Google and Opera. The main feature that WebGL brings to the Web is the ability to display 3D graphics natively in the web browser without any plug-in components. WebGL is based on OpenGL ES 2.0[5], and it uses the OpenGL shading language GLSL. WebGL runs in the HTML5's canvas element, and WebGL data is generally accessible through the web browser's Document Object Model (DOM) interface. A comprehensive JavaScript API is provided to open up OpenGL programming capabilities to JavaScript programmers.

As a technical detail, it is important to notice that the WebGL API is implemented at a lower level compared to the equivalent OpenGL APIs. This increases the software developers' burden as they have to implement some commonly used OpenGL functionality themselves. To make it easier and faster to use WebGL, several additional JavaScript frameworks and APIs have been introduced, including Three.js[6], Copperlicht[7], GLGE[8], SceneJS[9], and SpiderGL[10]. Such frameworks introduce their own JavaScript API through which the lower-level WebGL API is used. The goal of these libraries is to hide the majority of technical details and thus make it simpler to write applications using the framework APIs. Furthermore, these WebGL frameworks provide functions for performing basic 2D and 3D rendering operations such as drawing a rotating cube on the canvas. The more advanced libraries also have functions for performing animations, adding lighting and shadows, calculating the level of detail, collision detection, object selection, and so forth.

3 Lively3D: Host environment for web apps

The goal of the Lively 3D proof-of-concept design was to create a 3D environment in which applications of different kind – including data processing, visualization, and interactive applications in particular – can be embedded as separate elements within a single environment running inside the browser. Furthermore, the design is based on using facilities that are commonly used in the web already, implying that to a large extent it is possible to immediately reuse already existing content in the system.

[4] http://www.khronos.org/webgl/
[5] http://www.khronos.org/opengles
[6] http://threejs.org/
[7] http://www.ambiera.com/copperlicht/
[8] http://www.glge.org/
[9] http://scenejs.org/
[10] http://spidergl.org/

3.1 Overview

Web app, by simple definition[11], is an application utilizing web and [web] browser technologies to accomplish one or more tasks over a network, typically through [web] browser. Canvas application is a subset of web app, which uses a single canvas html element[12] as its graphical interface.

Lively3D[13] is a web application framework, where embedded canvas applications are displayed inside a three dimensional windowing environment. Individual applications embedded in the system can thus be composed using the Canvas API, offered by HTML5. In general, this enables the creation of graphically rich small apps that are capable of interacting with the user in a desktop like fashion.

The conceptual idea of Lively3D is based on previous project *The Lively Kernel*[5]. Lively Kernel was 2D window manager and IDE that was executed in the browser. Similar frameworks and tools have been developed by others like Ventus[14] and SproutCore[15].

The Lively3D framework itself is implemented with GLGE[16], a WebGL library by Paul Brunt, which abstracts numerous implementation details of WebGL from the developer. Embedding the applications to the framework was designed in such a way that the developer of a canvas application needs to implement minimal interfaces towards the Lively3D system in order to integrate the application within the environment. Existing canvas applications are easily converted to Lively3D app by wrapping the existing code to the Lively3D interfaces.

In addition to the applications, the 3D environment that displays the applications can be redefined using Lively3D interfaces. The applications and different 3D environments are deployed in a shared Dropbox folder, so that multiple developers can collaborate in implementing applications and environments without constantly updating the files on the server hosting Lively3D.

Lively3D is implemented as Single-Page Application (SPA) where the whole application is loaded with a single page load. This provides the user interface and the basic mechanics of 3D environments. SPA design was selected, so that applications can interact with the windowing environment and the whole state of the environment is stored within the JavaScript namespace. The design of Lively3D was considerably affected by the browser security model, which limits the possibilities of resource usage. The security model denies access both to the local file system and external resources in different domain with its Same-origin policy[17]. The policy is upheld in Lively3D with server-side proxies, so that the browser sees all the content in same domain. The main components of the system are illustrated in Figure 1. All components are designed with easy-to-use interfaces and require minimal knowledge of inner working of the framework.

[11] http://web.appstorm.net/general/opinion/what-is-a-web-app-heres-our-definition/
[12] http://www.w3.org/wiki/HTML/Elements/canvas
[13] http://lively3d.cs.tut.fi/
[14] http://www.rlamana.es/ventus/
[15] http://sproutcore.com/
[16] http://www.glge.org/
[17] http://www.w3.org/Security/wiki/Same_Origin_Policy

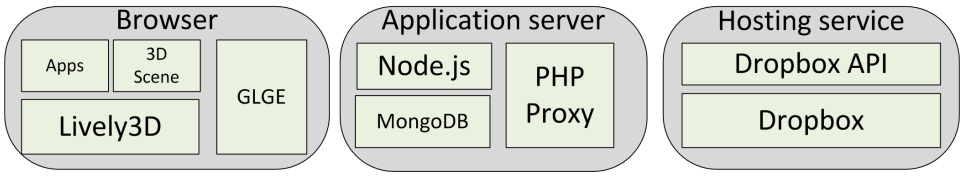

Figure 1: Structure of the Lively3D framework

Applications and 3D scenes are developed in JavaScript using Lively3D API, deployed to Dropbox using the official Dropbox client, and downloaded into Lively3D through PHP or Node.js proxies, depending on the situation. The Lively3D API provides resource loaders, which enable deployment of application and 3D-scene specific resources to the Dropbox so that complete applications and 3D scenes can be downloaded through the server hosting Lively3D, thus in essence circumventing browser security restrictions.

When a new 3D scene is designed and implemented, the developer has to define the essential functions that are called by the Lively3D environment, similarly to many other graphical user interface frameworks. These functions enable redefining how the system interacts with the user, including mouse interaction, the creation of 3D objects in the GLGE system that represents the application, and automatic updates of the scene between frames. Additionally, the initial state of the scene is defined in GLGE's XML format, which can be generated with 3D modeling software, like Blender (http://www.blender.org/) for example.

3.2 Lively3D apps

A Lively3D app consists of canvas application and its data structures in Lively3D host environment. Usable existing web apps are limited to canvas applications, because Lively3D is implemented in WebGL and the WebGL specification permits the use of canvas, image and video html-elements as the only source for textures within the 3D-environment. Most of the data structures are provided by Lively3D, but some conventions must be followed when converting existing canvas application to Lively3D app.

Since web apps are usually developed with expectancy that the app will be the only app in web page, the app structure can be pretty much anything the developer desires. But since Lively3D is implemented in Single Page Application paradigm, Lively3D apps are separated from each other with simulated namespaces as much as the browser model permits.

To achieve the above goal, each canvas application must have clearly separated initialization code. Additionally all the browser elements the app uses, must be created dynamically with a single canvas-element functioning as the only graphical element of the application. To mitigate these restrictions Lively3D offers API for canvas applications, which is presented in figure 2. In the following, we briefly list the most important features of the API.

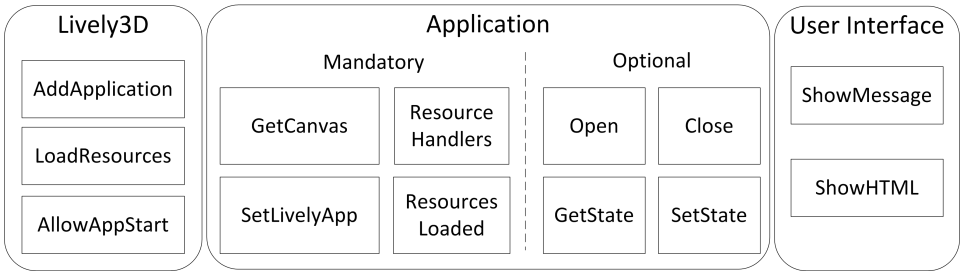

Figure 2: Lively3D API for applications.

To convert existing application to Lively3D app, the application must implement mandatory function of the figure. To embed the converted app to environment, the initialization code of the app must start the embedding process with calling the AddApplication-function. The process is presented in Figure 3.

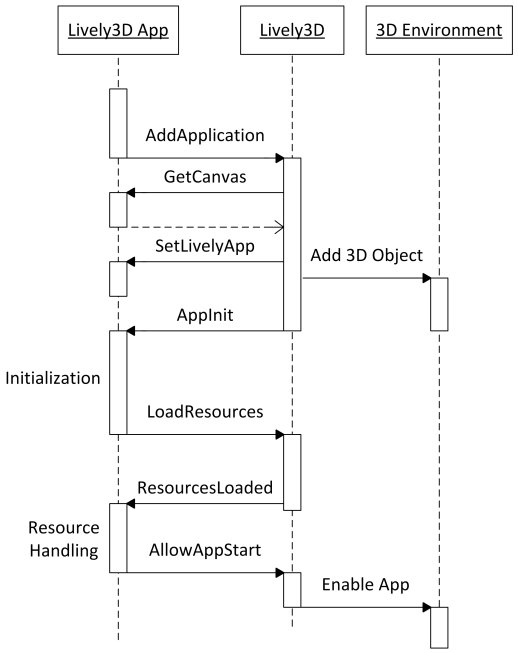

Figure 3: Sequence for embedding new Lively3D app.

As illustrated in the above figures, each application must implement a few mandatory functions and call Lively3D functions in certain order to advance the integration with the environment. During the integration, the canvas app is created and hidden with CSS-styling.

The Lively3D framework creates 3D objects representing the app and texturizes

them with the canvas element. Additionally to the mandatory functions, apps can provide optional functions which react to events like opening and closing the application within the environment. These function have default functionalty if they are unimplemented, but when the developer decides to provide them, they define what happens to the application status during the different events. Additionally, the inner state of the application can be serialized and de-serialized to developer's desired format.

Since the canvas element is defined as the only graphical element allowed for Lively3D Apps, the API also provides user interface functions to display messages and HTML in Lively3D provided dialogs. This provides consistent user interface, since Lively3D itself is rendered in a full browser window and possibilities of displaying text or other web interface elements within the environment are limited due to the WebGL specification. Figure 4 illustrates the existing canvas application in the left and the conversion to Lively3D app in the right with another app in the same environment.

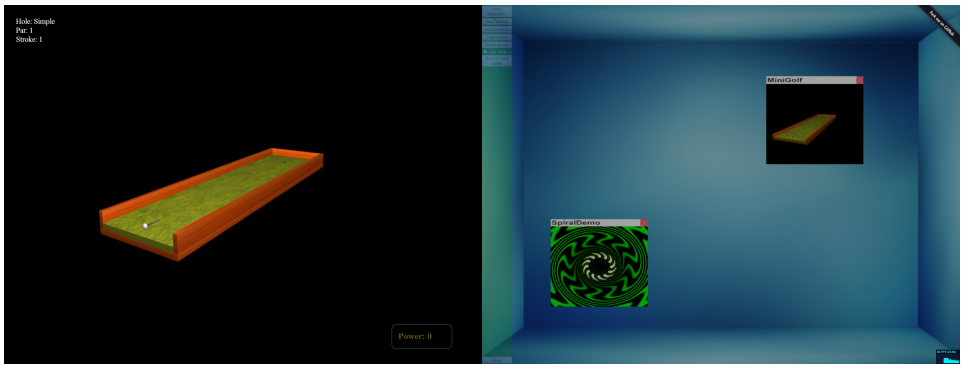

Figure 4: Conversion of existing application.

3.3 Redefining the 3D environment

As is common in various 3D applications, including in particular the genre of computer games, the visualization in our system is based on so-called scene graph, a generic tree-like data structure containing a collection of nodes. Nodes in the scene graph may have many children but most often they only need a single parent. In this structure, any operation performed to the parent is further propagated to its children. This flexible data structure enables numerous different visualizations, where the parent-children role can be benefited from.

The 3D environments in Lively3D are implemented dynamically, so that user can load new environments and change between them at will. As default only one environment is initialized in Lively3D and after adding more environments, the process of switching between environments is presented in Figure 5. Closing the applications and rebinding the events is done, so that the environment is in known

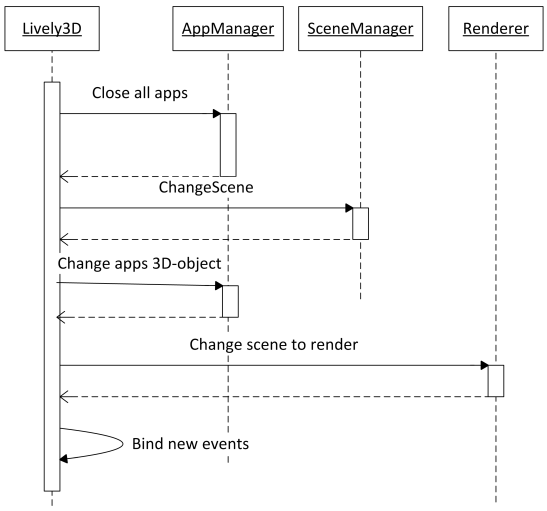

Figure 5: Sequence of switching environment.

initial state. Changing of the 3D-objects is required since GLGE allows 3D-object to be present only in one scene at a time.

In our experiment, we have created three different ways to visualize a scene graph where the children are applications and the root node is the 3D environment hosting the children. Example host environments include a conventional desktop, a planetary system where applications rotate a sun like in a solar system, and a true 3D virtual world, where applications move in a 3D terrain. These are introduced in the following in more detail, together with a set of screen shots to demonstrate their visual appearance.

Desktop. The conventional desktop consists of three dimensional room, cubes that represent closed applications, and planes that act as individual applications,

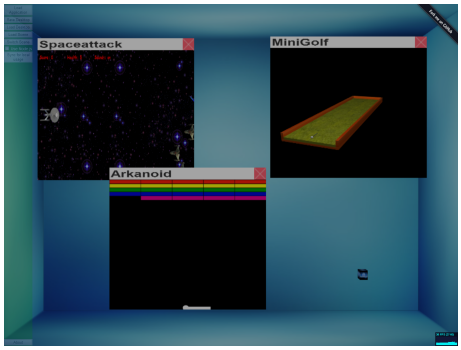

Figure 6: Visualizing the system as a conventional desktop.

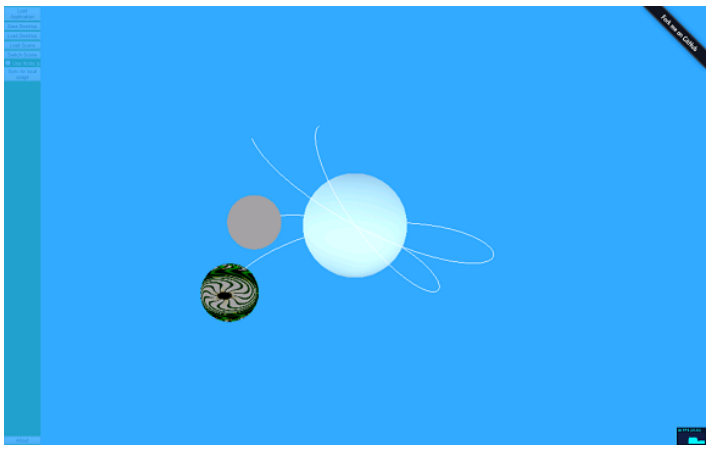

Figure 7: Visualizing the system as a solar system.

with the ability to execute JavaScript code, render to the screen, and so forth. A screenshot of the desktop environment, with three opened and two closed applications, is presented in Figure 6. The scene mimics all traditional desktop features, including dragging applications within the desktop and application interaction with opening, closing, maximizing and minimizing them with mouse controls.

Solar system. The solar system scene modifies the presentation of applications. In this scene, applications are presented as spheres that revolve around the central sun. Each revolving sphere generates a white trace in accordance to its path, and the trace is removed when the trace reaches maximum length. Each sphere uses the texture of the application canvas it is representing, and therefore each sphere has a different look within the scene. An example scene with 4 applications is demonstrated in Figure 7. Application windows retain their default functionality with dragging around, maximizing, minimizing, and so on. When an application that has been moved around is closed, the application returns to its position revolving around the central sun, in comparison to the conventional desktop scene where the application simply retains its current position.

Virtual world. The 3D virtual world scene goes even further from the conventional desktop. The only thing retained from the desktop concept are the application windows, and the only remaining controls for the windows are opening and closing the application, which then of course can introduce more controls within the application. The world itself consists of three dimensional terrain, where the user can wander around using the keyboard and the mouse. In this setting, applications are presented as spheres that roam the terrain in random directions, with their textures simplified to single image for performance reasons - experiences where application textures were used quickly showed that the resources of the test computer would no longer be adequate for such cases. Using this visualization, the 3D terrain and seven sample application spheres are illustrated in Figure 8. The right side of

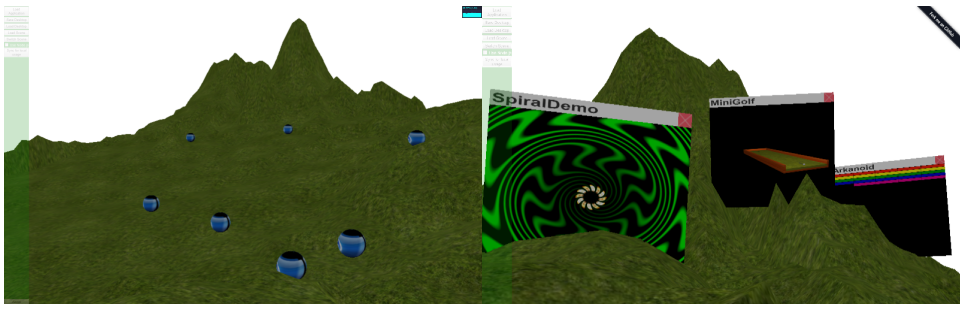

Figure 8: Visualizing the system as a 3D virtual world.

the figure illustrates application canvases within the world.

All of the above visualizations are based on the same JavaScript code, with the only difference being the rendering strategy associated with the scene graph. Consequently, in all of these systems applications are runnable, and can in fact run even when they are inactive and being managed by the different host environments, except when explicitly disabled for performance reasons.

4 Refactoring Lively3D UI

In this section, we introduce some early experiences regarding the relation between the Lively3D framework and widget libraries commonly used in desktop applications. To summarize problems, the original implementation was built directly on primitives emerging from WebGL, whereas the refactored version is geared towards widget libraries in its architecture.

4.1 Identified Problems

As a part of the process of designing the Lively3D framework, it became obvious that its architecture would benefit from more abstract programming concepts, in particular when considering the programming of the 3D UI. WebGL is a low abstraction level tool and 3D-engines building upon it only hide the rendering details from programmer. In particular such libraries lack essential concepts known from desktop application development.

As a concrete example, let us examine Lively3D's application window. The application window is a composition of three different 3D objects – title bar, window content and close button. These 3D objects are grouped together and aligned so that they appear as a window that is a solid object.

The background is that WebGL provides tools to create the 3D objects, align and group those, but there is no tools for creating a WIMP[18] elements such as titled window which can be dragged from the title bar and closed from the close

[18]Windows, Icons, Menus, and Pointer

button. However the natural abstraction of application window is an UI widget which has predefined look and feel, not a group of geometries which application logic is responsible for, which was the case in our original implementation.

Most of the 3D engines built for WebGL lack also necessary event handling capabilities. Using e.g. GLGE there is no way to bind an event listener to a 3D object. Determining which event happened and which object receives the event is responsibility of an application developer. In Lively3D the event handling for 3D UI is mixed into Lively3D application logic. In Lively3D there is a main event handler which catches all events for the Lively3D canvas, determines which object receives the event and executes functionality related to that object.

For instance, if the user clicks the close button of a window, Lively3D's main event handler will receive the event and calculate collision detection based on the mouse position to determine if the mouse hit any 3D objects. After finding the 3D object the event handler has to deduce which kind of object was hit and execute operations related to that object. In the window's close buttons case the operation would be to hide the group of 3D objects that forms the window.

In Lively3D there are only two kinds of 3D widgets – application windows and application icons – which receives only restricted amount of events so Lively3D has fairly simple 3D UI. However if we wish to add some new interactive 3D content to Lively3D we would need to refactor quite a lot of Lively3D code to get that done. Simple 3D UIs can be built using low abstraction level tools however the UI definition and logic becomes easily a mess of glut and glue solutions which makes it hard to maintain and develop the application further.

4.2 Revisiting the Design

Motivated by the above observations we created WebWidget3D, a 3D widget library for WebGL [3]. The idea of the library is to provide some predefined reusable 3D widgets and tools for building custom 3D widgets. WebWidget3D provides event system which enables binding mouse and keyboard events directly to 3D widgets. The framework also introduces predefined controls e.g. drag control and roll control which can be bind to any widget and fly control for moving camera in the 3D scene. The current implementation of WebWidget3D uses Three.js 3D engine for rendering, although 3D engine can be changed due specialized adapter component.

WebWidget3D provides predefined widgets and abstraction for creating widgets but it does not force the 3D world to consist of only 3D widgets. WebWidget3D content can be mixed with content (e.g. 3D objects, visual effects, animations, physics, etc.) provided by the 3D engine used with WebWidget3D.

We redesigned and reimplemented Lively3D's desktop UI using WebWidget3D to see how much refactoring would affect to Lively3D's complexity. The implementation is divided to two parts, 1) widget building blocks out of which complete widgets can be built (Table 1), and 2) a reduced set of ready-to-use widgets that can be used to create complete applications (Table 2). Figure 9 illustrates the revisited implementation.

Table 1: Building blocks of revised Lively3D design

Component	Description
GuiObject	Basic event handling capabilities.
Widget	Numerous commonly needed facilities for creating applications. Base class for new widgets.
Text	Simple string handling functionality.
Group	Abstraction of a container that can have other components as its children. Container can also have a 3D object representation.
Application	Corresponds to an application; receives events.

Table 2: Widgets used in Lively3D

Widget	Description
Grid window	Instance of a Group that is represented as a 3D grid plane. The grid window widget can be rotated in 3D space with the mouse.
Titled window	Instance of class Group. Contains three instances of Widget class as a title bar, a close button, and for representing the window content.
Menu window	Menu composed of multiple choice buttons. Individual choices are represented as cuboids.
Dialog window	Dialog composed of title text, multiple text fields and multiple action buttons.

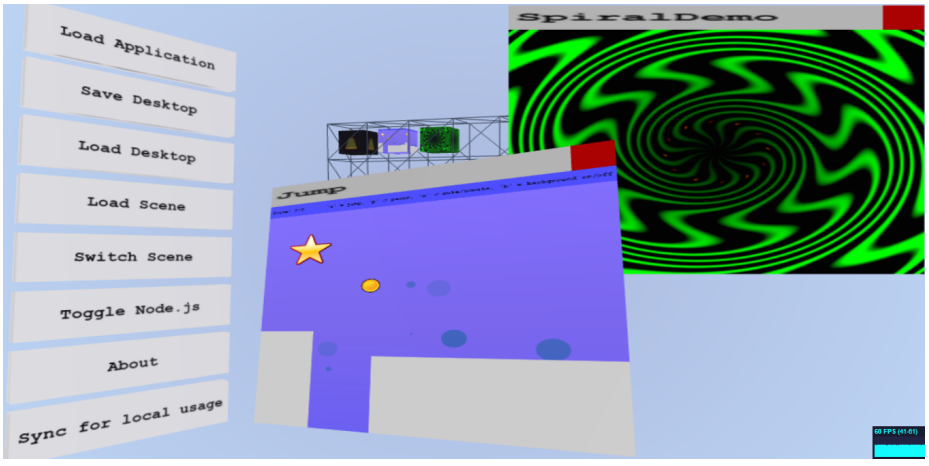

Figure 9: Reimplemented Lively3D.

4.3 Evaluation

In general, our work supports the conclusions of [2], where architecture related aspects of traditional applications are used as a driver for the design of apps that re run inside web pages. The goal of such designs, commonly referred to as single page web applications, is to support reuse and modifications in the long run, thus sharing the goals of more traditional software systems. Above, we reuse the design paradigm that is mature in the field of desktops, but to some extent missing from web design.

To evaluate the design, using WebWidget3D and its predefined widgets we were able to reduce the amount of JavaScript code lines by 26% [3]. In addition, using the library liberates developers to focus on solving application specific problems by allowing them to overlook numerous details that remain similar in different applications.

We also replaced Lively3D's 2D UI (menus and dialogs) with corresponding 3D UI widgets. This design reduced the number of code lines of HTML and CSS but on the other hand increased lines of code of JavaScript code.

5 Conclusions

Lively3D framework presents architecture to download and execute different applications within same environment. Although similar windowing environments have been developed and studied for years like Compiz/Beryl[19], our experiment runs on top of the browser. This approach has the advantage of the cloud, so that the user does not need to install anything else except the browser to execute the environment and the applications. This approach also works in different platforms from desktop Windows and Linux to mobile phones.

Our prototype demonstrates that integrating individual applications in a single web page is possible and achievable without complex structures from the application developer. However, one of the main goals – using existing content, preferably complete web sites in the system as applications – turned out to be unreachable. Due to the WebGL specification limitations, the use of existing content as textures is limited to image, video, and canvas elements, whereas in order to render existing web pages within 3D environment, the WebGL specification should to support IFrames as a source for textures. Currently, this option is associated with security issues - using the WebGL API gives loaded applications a direct access to the host devices hardware - which must be resolved before extending the rendering capabilities. Until then, applications are limited to the functionality of canvas element to produce graphics.

Additional security issues also emerge. Applications share the same JavaScript namespace which causes problems with variable overwriting. Even though each application has a simulated private namespace, variables might bleed through to the global namespace if the variable is missing var keyword. Applications can ac-

[19] http://www.compiz.org

cess global variables and overwrite them, including Lively3D namespace, other used JavaScript libraries and even browsers' default JavaScript functionality. This especially causes accidental problems with generic JavaScript libraries, since they are usually bound in $ variable, which is overwritten when new library is loaded and basic functionality of the environment brakes down as result. These problems could be fixed with proper process model where each application has its own private namespace and rendering context. There has been an emergence of JavaScript frameworks like Require.js[20] and browserify[21] that encapsulate parts of the JavaScript code to separate modules, this could be used as a pattern to fix some of the problems of Lively3D.

The Single Page Application paradigm has its advantages and disadvantages. Even though applications are in the same JavaScript namespace, this could be leveraged so that applications could communicate with each other. To enable this, the environment would need common JavaScript interfaces for application communications. Current implementation does not provide documented APIs for this.

One of the goals of Lively3D was minimal overhead code while embedding existing applications. We consider that this requirement was achieved quite well, although comprehensive analysis between converted applications is useless since amount of overhead code depends on coding conventions. In Lively3D most of the application initialization must be done dynamically in JavaScript code, as opposed to convential browser where HTML tags can handle some of the resource downloading. The minimal overhead code amounts to about 50 lines of extra code.

In the course of the design, we were alarmed by the fact that the circumvention of security restrictions became one of the key design drivers in the experiment. In this field, the problems arise from the combination of the current "one size fits all" browser security model and the general document-oriented nature of the web browser. Decisions about security are determined primarily by the site (origin) from which the web document is loaded, not by the specific needs of the document or application. Such problems could be alleviated by introducing a more fine-grained security model, e.g., a model similar to the comprehensive security model of the Java SE platform [1] or the more lightweight, permission-based, certificate-based security model introduced by the MIDP 2.0 Specification for the Java Platform, Micro Edition (Java ME) [4]. As already pointed out in [6], the biggest challenges in this area are related to standardization, as it is difficult to define a security solution that would be satisfactory to everybody while retaining backwards compatibility.

Finally, there are numerous new methodological issues associated with the transition. The transition from conventional applications to web applications will result in a shift away from static programming languages such as C, C++ or C# towards dynamic programming languages. Since mainstream software developers are often unaware of the fundamental development style differences between static and dynamic programming languages, they need to be educated about the evolutionary, exploratory programming style associated with dynamic languages. Furthermore,

[20] http://requirejs.org/
[21] http://browserify.org/

techniques associated with dealing with big data – data sets that are too large to work with using on-hand database management tools – data mining, and mashup development will be increasingly important.

To conclude, when considering the humble beginnings of the web browser as a simple document viewing and distribution environment, and the fact that programmatic capabilities on the Web were largely an afterthought rather than a carefully designed feature, the transformation of the Web into an extremely popular software deployment platform is amazing. This transformation is one of the most profound changes in the modern history of computing and software engineering.

In this paper, we are demonstrating the effect of new ways to visualize content in a fashion where the browser's new extensions are based on new web protocols rather than plugins, which has been the traditional way to create richer media inside the browser. Since no plugins that commonly introduce restrictions associated with their proprietary origins, the new technologies are manifesting the open web and open data. This, together with open data that is be available to everyone to freely use and republish as they wish without mechanisms of control, in turn liberates the developers to create increasingly compelling applications, building on the facilities that already exist in the web as well as their own innovative ideas.

References

[1] Gong, Li and Ellison, Gary. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2nd edition, 2003.

[2] Kuuskeri, Janne. *Engineering web applications: Architectural principles for web software*. Tampere University of Technology, 2014.

[3] Mattila, Anna-Liisa and Mikkonen, Tommi. Designing a 3d widget library for webgl enabled browsers. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 757–760, New York, NY, USA, 2013. ACM.

[4] Riggs, Roger, Huopaniemi, Jyri, Taivalsaari, Antero, Patel, Mark, and Uotila, Aleksi. *Programming Wireless Devices with the Java 2 Platform, Micro Edition*. Sun Microsystems, Inc., Mountain View, CA, USA, 2 edition, 2003.

[5] Taivalsaari, Antero, Mikkonen, Tommi, Anttonen, Matti, and Salminen, Arto. The death of binary software: End user software moves to the web. In *Proceedings of the 2011 Ninth International Conference on Creating, Connecting and Collaborating Through Computing*, C5 '11, pages 17–23, Washington, DC, USA, 2011. IEEE Computer Society.

[6] Taivalsaari, Antero, Mikkonen, Tommi, Ingalls, Dan, and Palacz, Krzysztof. Web browser as an application platform. In *Proceedings of the 2008 34th Euromicro Conference Software Engineering and Advanced Applications*, SEAA '08, pages 293–302, Washington, DC, USA, 2008. IEEE Computer Society.