Smorgasbord : A framework for increasing productivity in creating animations, simulations and games for use in online digital learning material

James Smith

MSc Computer Science project report, Department of Computer Science and Information Systems,

Birkbeck College, University of London, 2012.

Table of Contents

Abstract

Animation is a useful tool to use in a learning context, more sophisticated uses of animation include simulation and games. All three are considered to be an advantage of using digital/screen based media over print based media. The production of animations, simulations and games however is challenging and time consuming. A tool to simplify this production and increase the productivity of individuals creating this kind of learning materials is becoming necessary. The platform where this tool would be most effective is the web, to be able to be embedded in online learning content, to execute without complex installation on the users machine, to be run on various client platforms without alteration and to be available on mobile devices.

This project achieved the basic foundation framework for delivering web based animations, simulations and games, using a component system and incorporating a physics engine. Complex simulations can be constructed in a fraction of the time and development work is easily modularised, and reused.

The future for the project would be integration with an online virtual learning environment, a web based graphical user interface for authoring, and a community website for sharing code, knowledge and other assets. Ideas from the field of 'end user development' are seen as key to the future of bridging the gap between the creators of learning material and the skills necessary to implement them.

Supervisor: Professor Alexandra Poulovassilis

1. Introduction

The use of animation, simulation, and games in education has attracted a lot of research attention; developing theories and principles of when it is good to use animated content and when it is not, and principles in how to design good animated educational content [1.1] accepting that any form of media can be used for teaching and learning, if it is used correctly.

Educational animations have several advantages over static content, for example [1.2]

- Animations can show temporal change directly.
- Using animations instead of static graphics removes the need for added markings so that displays are simpler and less cluttered.
- Moving pictures are more vivid, more engaging and more intuitively comprehended.

Animation driven by program logic can be used to create simulation and games. The use of computer simulation and games in learning has raised debates on what the alliance of technology, education and media means. Researchers claim

- Computer games are both an opportunity and a necessity for the future of education,
- There is a "demand for teachers and schools to embrace new technologies, to make lessons more fun and to improve the 'slickness' of their presentations." [1.3]

Project Aim: This project aims to develop software to facilitate teachers, developers, educators and learning technologists in the production of animations, simulations and games as learning material.

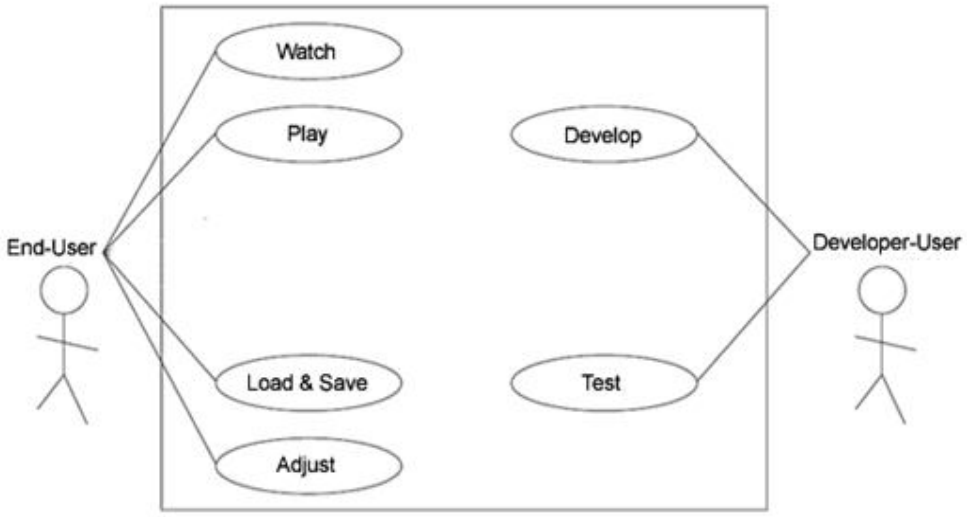
A category of software already exists for software designed to help in the creation and development of interactive applications with real-time graphical needs such as marketing demos, architectural visualizations, training simulations, and modelling environments. This category of software is referred to as 'game engines' [1.4].

Project Objective: The objective of this project is to develop a game engine for use in an educational context.

This stage of the project is primarily focused on creating a reusable framework which is extensible, flexible and with good performance to support the construction of learning materials by an academic developer or a learning technologist with some development experience. Ultimately this will form the foundation of a much more accomplished tool, which will contain an intuitive useable GUI, resource management and functions that allow content to be shared between authors.

2. Requirements

There are two users of our software, the 'learner', who is at the end of the process, and the 'developer' who uses the software to create the final presentation. I refer to these as the 'end-user' and 'developer-user'. I've outlined their roles in the use case diagram below.



2.1 Requirements Gathering

Requirement gathering was carried out with the staff of Goldsmiths University Learning Enhancement Unit (GLEU), a research and development department with responsibility to provide information and solutions regarding all aspects of technologies to support learning and teaching activities.

I have divided the requirements into 3 sections.

- Platform requirements - describe the necessary features of the platform the software will run on
- Functional requirements – describe the features of the software
- Non-Functional requirements – describe technical requirements which are not software features, such as performance.

In each of these sections, for ease of reference, each requirement is written after its argument with a bullet point.

2.2 Platform Requirements

Although there is no official audit in e-learning, a study by the HEFCE [2.1] revealed that:

- There are 510 web based courses offered by 113 HE institutions in the UK
- 73% of Open University courses use the web.

At Goldsmiths University, the Learning Enhancement Unit, who deal with e-learning in the institution, has recently been a hive of activity as their web based virtual learning environment (VLE) had to be significantly upgraded after it was singled out by the Warden (who is effectively their CEO) in his annual address as having potential for increasing revenue streams for the university, in what is becoming an increasingly competitive market.

Therefore it is important that:

- The system supports web based learning, i.e. it can be embedded into the web page in an online learning system.

With research interest picking up around 'mobile learning' [2.2] it also makes sense that:

- The system should be able to use a mobile phone as a platform.

In short, it should:

- Be available on as many devices as possible, mobile, tablet, desktop.

Following this requirement, if the software is to be deployed over the web, using the education providers IT infrastructure, the technical administrators will not want to expose their systems to unnecessary risks. Complex server installation would therefore be highly problematic [2.3]. Ideally, it also should not require any technical knowledge to install on the client. These considerations lead to:

- Simple to install for all users.

2.3 Functional Requirements

In order for the system to be flexible enough to be applicable to as many games/simulation cases as possible, I have taken recommended functional requirements from literature [2.4] [2.5]

- The developer-user should be able determine the graphics/sound by loading their own media resources
- Support for physics simulation
- Support for some artificial intelligence
- Support for 'character animation'
- Provide Sound/Audio – to provide effects and background music.

2.4 Non-functional Requirements

Performance of the animation system is the key non-functional requirement, In 'Computer Games and Software Engineering' [2.4] two requirements are stated that are relevant to game engines/simulators:

- Real-time animation. (the simulated motion per frame is linked to the actual real-world time elapsed between frames). Necessary for convincing physics simulation.
- Animation speed of 50 frames per second (at least 90% of the time).

3. Design

Our developer-user may be working with the software along with many other job responsibilities and perhaps limited time resources (for example, our developer-user may work in a university department with a tight budget). Inconsistent client software, physics simulation and platform optimisations are areas of specialist knowledge that are time consuming to keep up to date and we cannot assume our developer-user user has the time or motivation to learn them.

Ideally we want to leave our developer-user with nothing more to worry about than the details of their current project; our software should be designed to help with this problem.

If the client side software changes, our game engine should be able to change, as much as possible, without disturbing the code written by our develop-user, avoiding rewriting thus saving time.

Program Efficiency vs. Programmer Productivity is a trade-off central to the design of a game engine. Some commercial game engines [3.5] include a domain-specific language (DSL), whereas some are simple application libraries that the developer-user "links" to their project and thus can use the objects/functions/methods [3.6]. In between those lie various levels of 'frameworks'. A framework is a re-usable software system or subsystem with functionality to help develop and glue together the different components of a software project [3.7], various parts of the framework may be exposed via an API.

3.1 Comparison of DSL, Libraries and Frameworks

DSL	• More expressive than general programming languages • Functionality limited to specific application domain • Productivity gain from expressiveness • Reduced maintenance (as usually more expressive languages create less code)
Library	• Less indirect execution than DSL gives better performance • API can provide a domain specific vocabulary helping the expressiveness and ease of use • Simpler mapping to functionality which lies outside the domain, things like error handling functions are difficult in a DSL
Framework	• Less indirect than DSL but often key objects/data-structures referenced through pointers to enable necessary flexibility, so more indirect than using calls to a library. • As with libraries, the API can provide a domain specific vocabulary helping the expressiveness and ease of use. • Functionality to automatically create the architecture increases programmer productivity.

DSLs require more development; the application must perform validation, parsing and generating of the executable code. A standard solution using XML has become popular for some DSL solutions, this way using XML schemas to define the language the validation and parsing can be dealt with by existing software libraries. There are drawbacks to using the purely declarative XML syntax however, algorithmic routines become verbose, for example a simple iteration loop would likely look something like this (pseudocode) :

```
<for var="i" val="1">
 <if var="i" op="<" val="10">
  <plus var="i" val="1"/>
  <call methodname="doSomething"/>
 </if>
</for>
```

For this reason, XML based languages tend to be used for cases where there is no need to define algorithms, such as declaring user interfaces (MXML), defining services (WSDL), configuration files (ANT) or page layout (XHTML). In our current project we are mainly trying to increase the programmer's productivity in creating the logic of the animated behaviours so XML is not a good choice, but in any cases where we need configuration it is worth consideration.

Our aim to support the develop-user as much as possible means it would be desirable to free them from concerning themselves with building a software architecture, from that we conclude that a simple library would fall short of our hoped for improvement in programmer productivity.

For the above reasons; effectively achieving our aims balanced with cost-effectiveness (even if purely measured in time) in both build and maintenance, a framework with a well designed API, is the most feasible and desirable choice.

3.2 Defining the Framework

Frameworks which abstract over a complex, general or changeable API to provide simpler, more domain specific functionality are commonly used in software development companies, for example in mobile app development, the iPhone hardware interface portion of an application may be written once as a reusable framework, and after this there is no need for future projects to touch the underlying APIs, the framework will do that for them, this framework may in fact, have been developed by someone else and bought in.

Realistically, the more the framework abstracts for you, the more specialised it becomes, so, for example when you write a game engine, it is a specialisation, and there will be projects it is not suitable for.

As a framework develops, and more is added to it, it 'converges' on a set of limitations. Conceptually, they can be pictured as a pyramid - the base is wide and supports the rest, while the top is a focused point.

In the case of game engines, a highly specialised framework allows only a very small set of games to be built on top. If the top half of the pyramid is removed, there will be a much wider base to build a larger set of games on - those games will, however, need to have more building work as the framework is not providing as much. [3.2] The design needs to find the balance between specialisation and productivity.

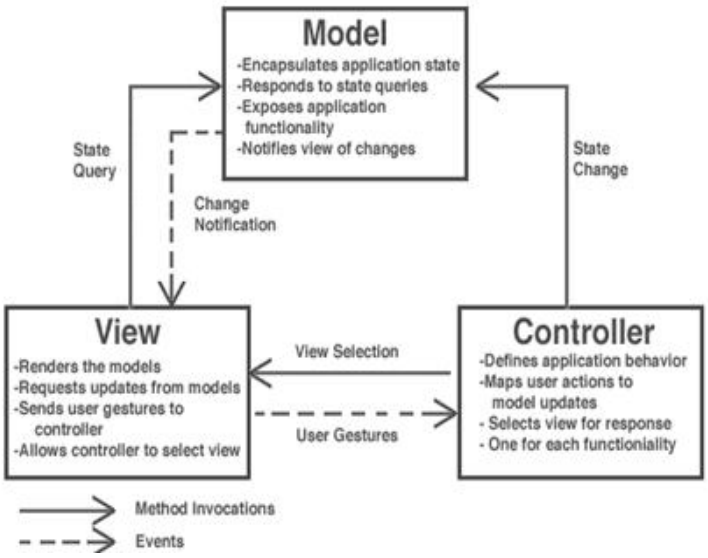
3.3 High Level Architecture

Generally, every component in a game engine can be classified it as belonging to one of three primary categories [3.1],

1. the application layer, dealing with the hardware and the operating system
2. the game logic layer, managing the game state
3. the game view layer, presenting the game state with graphics and sound

The use of a separate module for the view layer should enable us to choose between the various rendering techniques, giving us graphics system independence, helping us to cope with the possible changes and new features in the rapidly changing web client landscape – e.g. we can use a different animation rendering module optimised for an iPhone browser, or swap this for a rendering module which is optimised for a Windows desktop system - all of which should not involve changes to individual projects developed by our developer-users, which should be confined to the game logic layer.

This points towards a 'Model-View-Controller' like architecture. A Model-View-Controller architecture is designed to keep the data model and the functional core that manipulates the data independent of the user interface. [3.4]

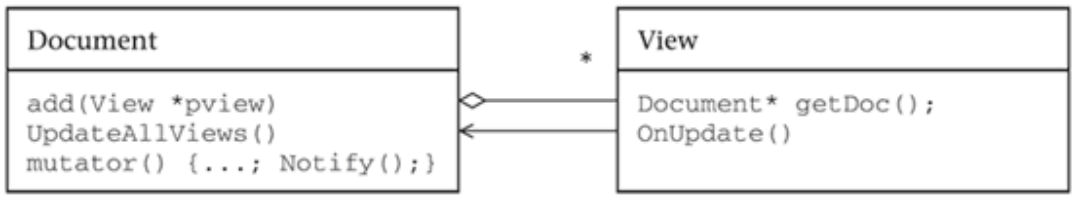


In our game engine, the 'view' is both the animated output and user input from the mouse, keyboard or touch-screen, the data model is the 'simulated world' and contains the state of all of the objects currently in the world, and the controller contains the behavioural algorithms of the objects in the world.

This architecture, in a game engine, violates the principles of encapsulation and responsibility of object-orientated design. Unlike many applications, the data model is composed of the same objects that we want to define the behavioural algorithms in. There is no separate structured 'data source'.

A simpler two layer architecture, known as the Document-View architecture is more suitable. A Document-View architecture recognises all three roles of the Model-View-Controller but merges the three tiers into two where appropriate, in our case, the Model and Controller layer. The architecture will consist of broadly two layers:

- The document: holds the data involved in your game: the characteristics and positions of the game objects as well as the behaviours of the game objects.
- The view: the graphical object in charge of displaying your window on the screen.



The Document-View pattern

3.4 Naming Convention

As we approach the boundary of conceptual architecture and implementation, in order to avoid confusion between the objects in our game world, and objects in the sense of programming language constructs, we can define some new terminology for our domain model.

The English version of the Swedish word Smorgasbord has come to mean:
"an extensive array or variety presented together" [3.8], this provides us with a nice name for the framework. In this way the game world is the 'bord' and the variety of objects within it are the 'smorgs'. It also has the added bonus of being a fun word to say.

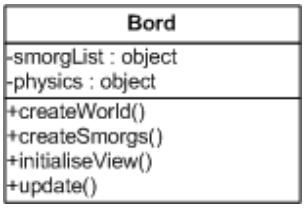
3.5 Design of the Document Module : The Bord

A game engine is a kind of 'simulator', (in a loose sense) – capable of simulating basic physical reality, or bringing to life a more abstract world.

This notion of a 'world' is a key concept; the 'world' will define the rules, e.g. gravity, momentum, collisions. The 'things' in that world will obey the world's rules. The 'world' therefore has a hierarchical relationship to the things within it.

The architecture of the world (bord) will reflect this by 'containing' a list of objects (smorgs), this list will contain an extensive variety of objects, each capable of responding to those rules in their own way. This a text-book example of the object oriented technique of polymorphism.

A initial design for the Bord is illustrated with the UML class diagram below.



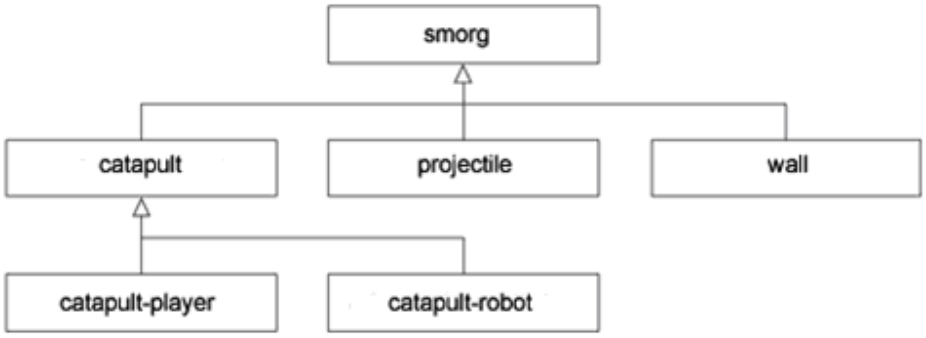
3.5.1 Simulating Physics

In line with the prinicples of object oriented design, whereby no object should do too much, I've chosen to delegate physics simulation to another object, this also brings the additional possibility of leaving the physics code out of the world if it is not necessary.

3.6 Design of the Game 'Objects': The Smorgs

As mentioned, the bord module will hold a list containing an extensive variety of smorgs, these will be polymorphous objects, that is, the bord module will treat them in a uniform way, for example, by calling an update method on each frame, and each smorg will respond with behaviour appropriate to its type. One strategy for implementing this polymorphism is to use inheritance to differentiate the smorgs into types.

Type	**Example behaviour**
wall	Does not move, May cause lifespan of other objects to decrease if hit at sufficient speed.
projectile	A smorg that is launched by another smorg
catapult	A smorg that can launch a projectile
catapult-player	A catapult smorg that is under control of the player
catapult-robot	A catapult smorg that is under control of the game AI



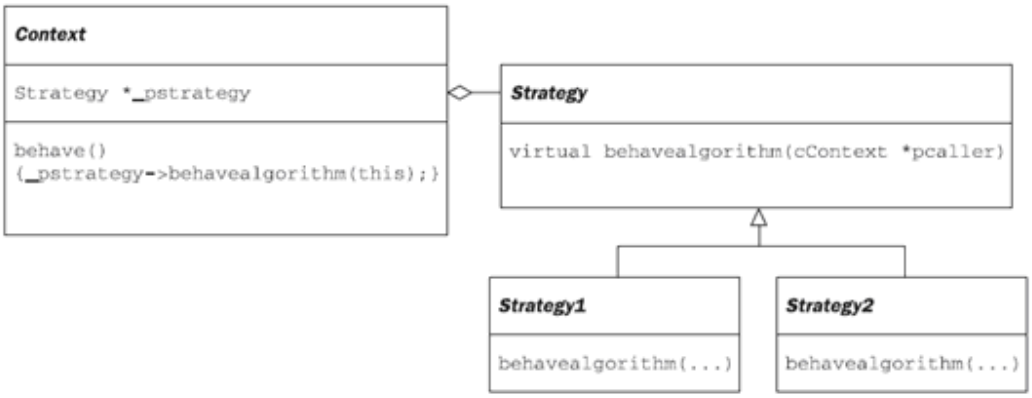
(Class diagram showing inheritance of smorg types)

The Smorg class itself would likely be an abstract, base or interface class which defines the methods which the subtypes override. The set of methods to override is in important design decision, as it will influence the API that the developer-user works with (more on that topic in the next section).

However, continuing to use this kind of inheritance will unfortunately lead to unmaintainable code, suppose, for example we want a smorg which is both a wall and a catapult, we would need to inherit from both catapult and wall, this kind of multiple inheritance is not supported in many languages (e.g. Java, C#) or not recommended as can become difficult to predict effects of code changes. It will also lead to a 'combinatorial explosion' [3.3] for example if we have a behaviours for flocking, obeying gravity and bouncing, we have to implement all those behaviours within the class heirachy – so to implement those 3 behaviours for 3 parent types, we have to create 9 types of object.

3.6.1 Design Pattern: Strategy

A better design is to use composition to differentiate the smorg types. This means that a set of behaviours is implemented in their own classes and 'plugged into' the smorg type at runtime, this overcomes the combinatorial explosion problem imposed by the type hierarchy, and also will enable more dynamic behaviour as, unlike with inheritance the plugged-in classes can be swapped in and out during run-time.



Determining the balance between inheritance/delegation and determining the API to configure the 'strategy' behaviours is another important design decision. I have chosen to create two kinds of strategy classes,

1. Listeners - this type will be responsible for responding to I/O and system events and
2. Forces - this type will be responsible for responding to information from the physics code

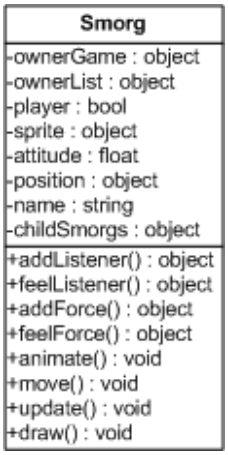
An example of using the delegates would therefore be something such as:

```
example->addListener(mouseEvents, params);
```

and implementation of feel listener would look something like this:

```
Smorg::feelListener(listener)
{
    switch listener:
      mouseLeft://etc
      mouseRight:// etc
}
```

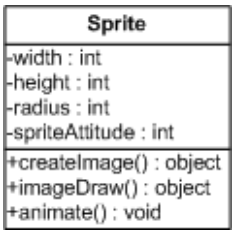
An initial design for the smorg object is shown in the UML class diagram below.



The boolean 'player' should be true if there is one smorg in the list that represents the player, this can be handy as the player's representation in the world is often treated differently to other objects - for example if the world is larger than the screen/viewport, the portion of the screen displayed is usually centred on the player.

In this initial design I have chosen to represent the position and attitude smorg in the world using a combination of a coordinate vectore and a floating point value called attitude which will be the angle the smorg is currently turned through, starting at zero for the initial attitude. Later when considering the possibility of having either a 2D or 3D view of the world I will probably swap coordinates with angle for a single matrix representation using a normal and tangent vector, adding a binormal for the 3D case. This design decision is purely based on a combination of time budget and my (regretfully) less than full grasp of linear algebra (at this time).

The other important delegate of the Smorg object is 'sprite' property, this is a very platform dependent class, and is responsible for handling the appearance of the smorg in the game world. A initial design for the sprite object is shown in the UML class diagram below:



The animate method is included for sprites with multiple frames, either representing direction, movement or some other state.

3.7 Design of the View Module

The view module is responsible for rendering the state of our world, creating a visual image on the screen. In this phase of the project 3D graphics are too complicated (time consuming), so we will be concentrating on creating 2D graphics, although designing a architecture flexible enough to allow 3D graphics to be added.

A 'graphics pipeline' consists of every process that happens to take the game world data and use it to create the final image on screen. In total it includes both hardware and software, in this project we are only

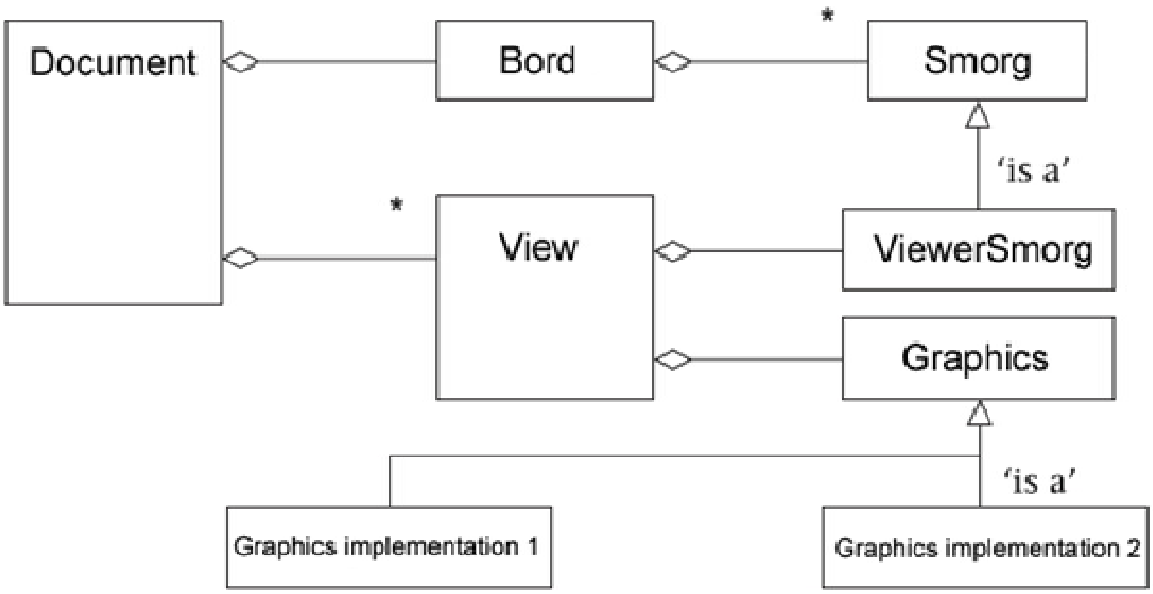
concerned with a section of the software portion. The client machine's operating system and hardware drivers take care of the low level details, so we concentrate on the higher level functions.
A 2D pipeline consists of taking the coordinates of each of our smorgs and applying rotations and translations to them depending on the relative viewpoint which we have chosen to look into our world in. In two dimensions, this means knowing what is the X and Y, rotation and zoom of the viewpoint,

Because the viewpoint itself has an x, y position and orientation in our game world and also we may want our viewpoint to behave according to the physical rules of the world (obey collisions, or other constraints) – it makes sense to think of the viewpoint itself as a special kind of smorg.

The final step in our pipeline is to take the bitmaps or other visual information from each smorg, scaled appropriately, and copy them at the correct positions into a final single image. The final step is the most 'implementation dependent' step, and there are many potential platform dependent ways we may want to provide possibilities for this, for that reason the final image composition should be another 'strategy' class that can be easily swapped into the framework.

3.7.1 Design Pattern: Bridge

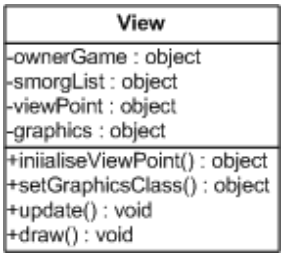
To maximise flexibility we can use the bridge design pattern. The bridge design pattern uses composition (delegation) to decouple the implementation from the interface.



View will be implemented much like a Java interface type, it will not implement any of its own methods, each method in the interface still has to be implemented, but instead of the implementation being in the subclass, they will all be implemented by delegate classes.

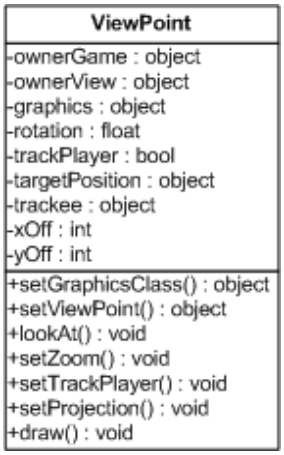
We can now create a set of 'View' interfaces (e.g. 2D, 3D) and a set of 'Graphics' implementations (e.g. bitmap or vector for 2D / raycasting or raytracing for 3D) rather than permanently binding our implementation to the interface. We loose the compile time ability to check that our implementation classes really have implemented each method in the interface, but provided we fulfil this, we can still program against a known (set of) interface(s) without worrying about how it is implemented.

An initial design of the view class is shown in the UML class diagram below:



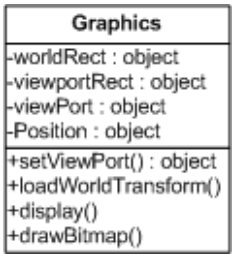
3.8 Design of the ViewPoint Module

ViewPoint an object which inherits from Smorg, but without an sprite of it's own, is responsible for acting like a 'camera' in the world, it's position and angle information are fed into the rendering pipeline, each of the other Smorgs in the world have their position translated by the inverse of the different between them an the camera, if the camera moves up and left, all the Smorg images should be translated down and right, if the camera rotates clockwise, each of the Smorgs should be translated and rotated appropriately.



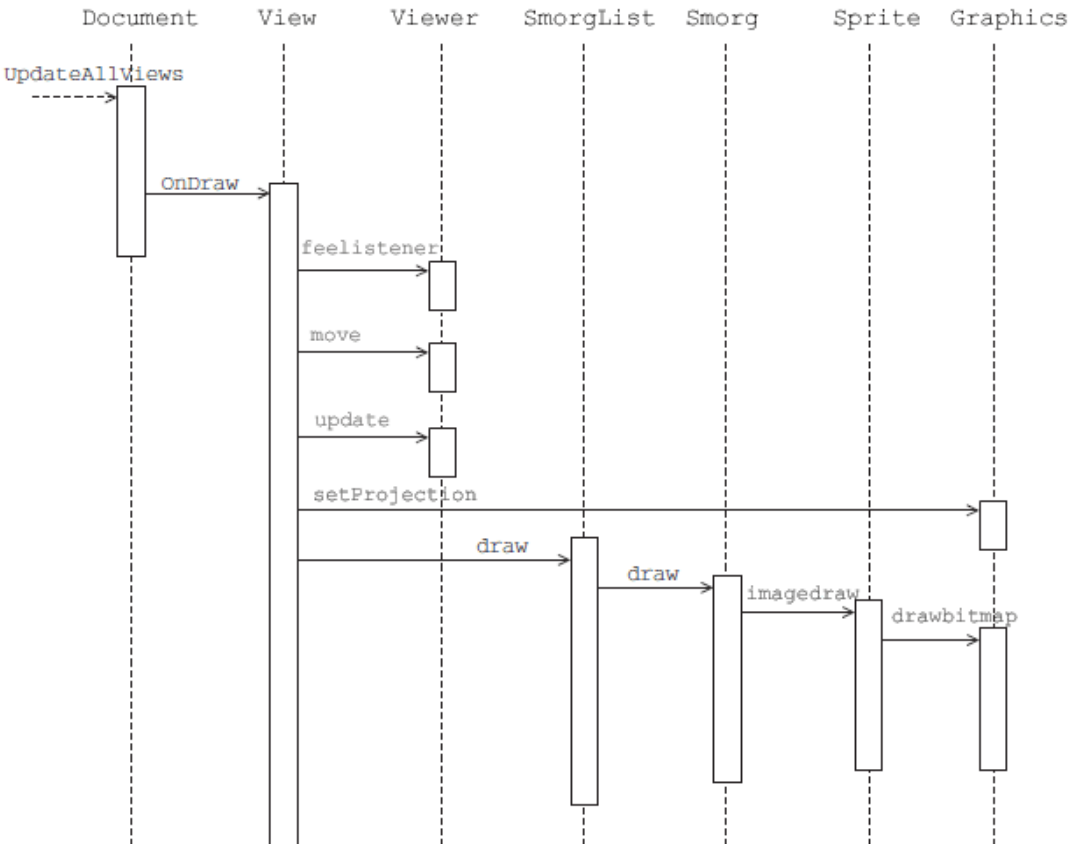
3.9 Design of the Graphics Module

The lifetime of the Graphics Class class after initialisation, is a definition of the rendering pipeline, and is very dependent on the platform we will be running on. A initial design is shown in the class diagram below:



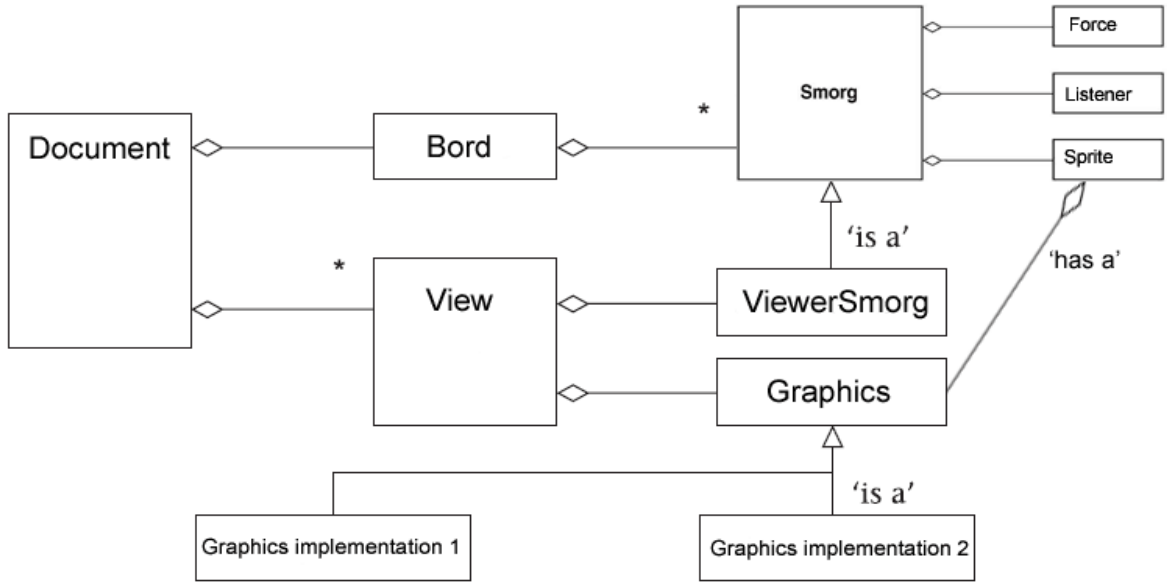
3.10 The Rendering Pipeline

The full rendering pipeline is shown in the UML sequence diagram below:



3.11 Summary of the high level architecture of the Smorgasbord framework

Putting these classes together gives us this high level view of the overall architecture:



(UML Class Diagram of High Level System Modules)

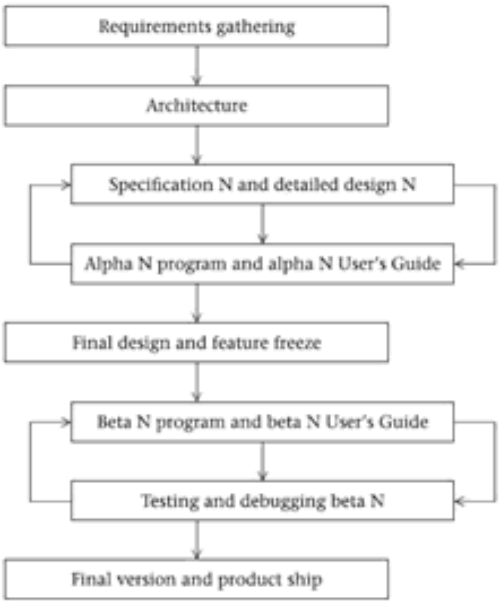
4. Methodology

4.1 High-Level Methodology

Time constraints on this project are severe in comparison to many real-world software development projects. Although any software project cannot be risk free, a strategy of risk reduction for short projects is even more important. The staged deliver model [4.0] was born out of experience of managers at Microsoft and is most often employed in rapid development projects. Like it's parent methodology Agile, it is iterative, but has refinements.

The development process is broken into alpha and beta stages, within each it is expected that there will be numerous short build/test cycles. In the alpha stage, the software features of the final product do not have to be set, allowing for changes in the features specified, running concurrently with build/test cycles, responding to feedback. This gives a good indication of the length of time for final build/test. When the remaining project time gets to the point where there isn't enough time for 2 alpha stage builds, the feature set is frozen and the beta phase begins, and the focus shifts to developing the most robust and bug free version possible.

A flowchart of the process is shown below:



Development of the user guide is concurrent with the development of the software.

4.2 Low Level Methodology

Behavior-driven development (BDD) is a strongly recommended development methodology to framework code production. It is an approach that adds a semantic layer to unit tests used in test-driven development which aids both in the design of tests and the relationship of the tests to the user requirements. Tests cases are written in natural language that a non-programmer can understand and are named with whole sentences beginning with the word 'should' e.g. shouldDetectOuterBorder or shouldImportBitmapFromFile. The idea is to create well defined outputs and deliver thoroughly tested software [4.1]

5. Implementation

In order to deliver richly interactive content to a web client you need to deliver some kind of executable instructions. Ideally the same instructions can be sent to any supported platform and it will execute in roughly the same manner. On the web there are 2 main strategies for achieving this:

- Ahead of time (AOT) compilation to byte-code which is served by the webserver then executed on a virtual machine installed on the client (e.g. Java, Adobe Flash Player)

- Just in time (JIT) compilation of source code (served by the webserver) by a compiler installed on the client machine (e.g. ECMAScript/JavaScript)

5.1 Choice of platform

I considered the 3 main technologies with regard to the use of these strategies on the web currently.

- Java – mainly AOT compilation to bytecode for a virtual machine (JVM)
- ActionScript – also AOT compilation to bytecode for a virtual machine (AVM2)
- JavaScript – JIT compilation by a compiler embedded in the web browser. (v8, Tracemonkey)

Firstly it's important to say that all each of the current technologies employ both strategies, Oracle's Hotspot, a Java Virtual Machine, uses JIT technology (mainly inline caching) [5.4] as does Adobe's AVM2 embedded in Flash Player version 9 and upwards [5.5]. Google has created Closure a Javascript-to-Javascript AOT compiler that performs static analysis, creating optimised source code for final delivery to the browser, so there is usually a blended approach to the strategies.

The benefit of AOT compilation is that the compiler has enough time to perform static analysis on the source code which enables it to make optimisations, this is a time luxury that the JIT compiler does not have. There are however, problems with the 'AOT bytecode to virtual machine' approach when used on the (potentially hostile) web.

Java's security is complex, because the delivered byte-code is at a lower level, in comparison to the JavaScript source code it is harder to verify for safety, according to Prof. Michael Franz of UC Irvine, the JVM's byte-code verifier has O(n^4) complexity [5.6], making it slow and error prone, this is why Brian Krebs, computer security and Washington Post columnist named the JVM 'the number one paid for malware vector', in 2010 Microsoft described the situation as an 'Unprecedented wave of Java exploits hitting users', and security experts Symantec added 'Java makes a tempting target [for hackers]'[5.1]

Leaving many to conclude: JavaScript is as low-level as a web programming language goes [5.7].

In addition to this, Java start-up time is often slow the slow start-up time is due to IO-bound operations; the rt.jar class data file alone is 40 MB and the JVM must seek a lot of data in this huge file [5.2].

Of the 2 strategies, delivering source code to a client for JIT compilation looses on run-time speed but wins out on start-up speed and security.

The question then is: which one of these factors is most important? To answer that we can look at performance benchmarks for each technology performing operations similar to what we are aiming to do.

Fortunately bubblemark.com [5.8] does exactly this, a benchmark for browser based animations. Running the benchmarks on an Acer Aspire One laptop with 1.66Ghz processor, 1GB Ram, 512 KB L2 cache, 64 MB of dedicated video memory I got the following results:

- Java (Swing) = 181 frames per second (fps)
- Flash Player 9.0 = 57 fps
- TraceMonkey JIT compiled JavaScript in Mozilla Firefox = 57 fps
- V8 JIT compiled JavaScript running in Google Chrome v20 = 171 fps

The requirements for animations created with this software is they should play at roughly 55-60 FPS, which means that any of the technologies will do, so I am discounting Java – I do not need the higher performance and see no reason for increasing potential security problems on the end users machines, it is also unavailable on Apple iPhone and iPad platforms. A roughly equivalent argument about security and Apple support can also be made against choosing Flash player as the target for this software [5.16].

Thus, JavaScript is the target language, and the various browser embedded JavaScript JIT compilers are the target platforms.

One of the most interesting questions about this choice for implementation is: how does this project get implemented in JavaScript?

This question may not immediately seem of interest, but JavaScript was designed as a lightweight interpreted scripting language for non-professional programmers, it is based on Scheme, Smalltalk, and Self but given a java-like syntax at the last minute, a decision made by the marketeers at Netscape (where the language was invented) not by the engineers, where, the legend has it, it was developed in 10 days [5.9].

Since 2007 with the release of Google's Chrome web browser, the role of the language has changed. V8, Chrome's JavaScript compiler was designed by Lars Bak, who worked on the Self programming language since 1991 and is a specialist in JIT compiler design [5.10]. He introduced optimisation methods such as type inference, inline caching and generational-incremental garbage collection which have improved performance by at least 10 times in benchmark tests. Now serious attention is being given to applying software engineering methodologies to JavaScript projects.

The challenge with this is that Javascript is not a strongly typed, class based imperative language, and much software engineering methodology has grown up with C++ and Java which are. The result is, the use of established design patterns and techniques may not be as obvious when implementing in this language.

5.2 Implementing Object Oriented Programming in JavaScript

Object oriented design is currently the most widely used methodology for software design; programmers are very familiar with the concepts and notation. For this reason it makes sense to use an object oriented programming style.

JavaScript is based on a prototypical (Self) and a functional language (Scheme); even though it has an added 'cloak' of Java like syntax. It is more declarative than Java and C++.

In JavaScript 'basic' objects are simply associative arrays, (sometimes referred to as objects with slots). The language includes several more specialised built-in object types, the most important of which is the Function object type.

A Function object adds a 'call()' method, a 'body' property, and an 'arguments' property. The call method executes the list of statements stored in the 'body' property, using the values in the 'arguments' property. The list of statements is set when the function is declared.

For example:

```
function myFunc(arguments) {
        /* statements */
}
```

Can be thought of as syntactic sugar for something like:

```
myFunc = new Function(arguments, { /* statements */ })
```

i.e. - The declaring of a function automatically creates a new instance of the built in Function object type.

This gives JavaScript 'first class functions': the language supports passing functions as arguments to other functions, returning them as the values from other functions, and assigning them to variables or storing them in data structures [5.11].

In the 'Strategy Pattern', which is the design pattern we chose for implementing the smorgs, aggregation is used to determine the functionality of the object. The behaviour of the smorg is determined by plugging behaviours in to the smorg object. With first class functions, this is a simple matter of assignment. JavaScript programming is fundamentally building objects by aggregation.

5.2.1 Using Prototype Chains for Inheritance and Code Reuse

In C++ and Java, objects are defined in a 'class', a construct which is then used to create instances of the objects. A class defines the structure and behaviour of its instances, instances hold the local data representing the state of each object. Classes can be extended, allowing code reuse, by an inheritance mechanism which allows one class to extend another, in a hierarchical sub-class (extender) / super-class (extended) relationship.

In JavaScript there are no classes. JavaScript is based on an alternative model known as prototype-based programming. Prototype-based programming was a highly active research topic in the late 1980s, and numerous papers on the topic were published in the ECOOP and OOPSLA conferences in the late 1980s and early 1990s. [5.17] The notion of class was rejected by promoters of prototype-based languages because they wanted to be able to program by directly creating and manipulating objects as it seems to be the most natural way of thinking for humans. [5.19], [5.20]

In a prototype-based language, an object holds its own description, it can be directly created, using a literal syntax, and exists in a mutable state. The idea is to create a simpler system, the programming model of prototype-based languages is based on one basic construct: objects with slots and a few mechanisms: cloning and delegation.

5.2.2 Advantages and disadvantages to the prototype approach

- The advantage in class based systems is that the class represents both a description and a conceptual set, an objects 'belongs' to the class which instantiated it. This gives objects strong identities (types) and provides a level of abstraction to help manage complexity, for example with UML class diagrams, which give a static conceptual overview of a object system. This helps the programmer organise large projects as is arguably a keystone in the success of Java, C++ and the modern industry of object oriented software engineering. In the prototype system there is no support for this kind of organisation in the language.

- The disadvantage in class based systems is that the classes are abstractions of objects, they represent 'concepts' in the domain being modelled, therefore they are identified early in the design phase of the software. If new classes have to be introduced to the software after the build phase there may be considerable impact on the classes, the inheritance hierarchy may have to be restructured, some classes may disappear or be merged due to the introduction of more or less specific classes – this may lead to extensive refactoring, or a complete return to the design phase which is natural place for classes to be created.

- Class hierarchy changes and reclassification of instances are not simple in class-based systems, however this is straightforward in prototype-based systems.

5.2.3. Examples of Prototypal Inheritance in Practice

The cloning and delegation mechanisms in a prototypical language provide two ways of achieving inheritance (code re-use).

Cloning must be combined with concatenation, adding in the new functionality of the extended object– this more mirrors ideas found in nature, where inheritance is achieved primarily through cloning.

Delegation is achieved by the use of composition, whereby objects hold pointers to other objects. This strategy can be achieved in classical languages too, but is made much easier by the dynamic nature of the prototype system objects which allows concatenation of new members as well as swapping of existing ones.

JavaScript supports both cloning and delegation as strategies for inheritance, it does however strongly favour delegation. It incorporates a specific pointer in each object which is automatically followed during member look-up. If, for example, a method is called on an object and it is not found, the system will automatically follow this pointer to the delegate object, and if it still not found the special pointer is followed from this object to the next one. This means that objects form a chain, or heirachy, in which the delegate upward. This idea was originally pioneered in the language Self.

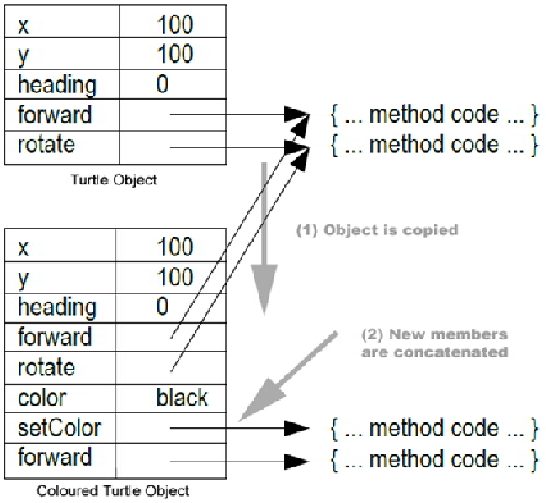
A popular example scenario for illustrating these concepts is a simple model of a Turtle graphics system similar to the one used in the Logo programming language [5.21][5.22]

In this example we look at two objects in the system:

A monochrome turtle – contains the integer members x and y that define it's position and a heading which defines it's orientation. It also contains the methods forward() to move the turtle forward and rotate() to change the orientation.

A colour turtle – has everything that a monochrome turtle has but extends the object with a colour property and a setColour() method. It also overrides the forward() method to a method that enables colours to be swapped during the drawing of one line

5.2.4 Implementing inheritance via copying



This is a 2 step process:

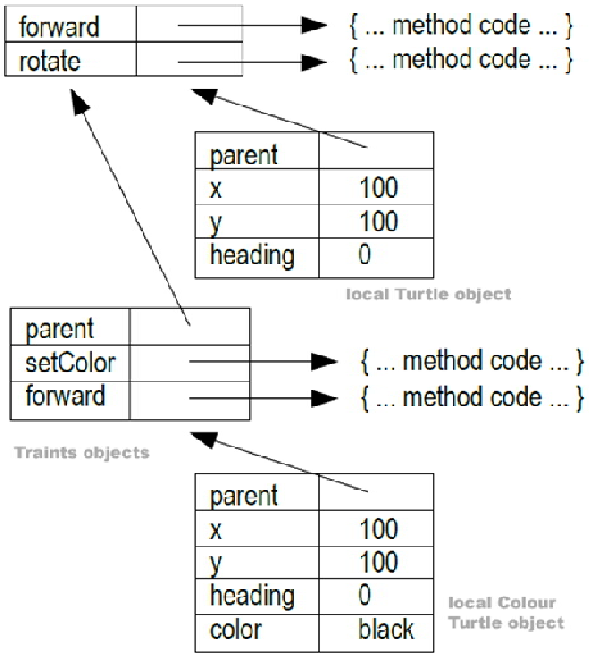
1. Copying the turtle object
2. Concatenating the new members.

In this scenario the only issue is to deal with the overridden 'forward()' method, which can be achieved by literally overriding the old function pointer, or by masking it by giving the newer forward precedence.

The coloured turtle object does not maintain any link to the turtle object

<Diagram source: Antero Taivalsaari, (2009) ref 5.17>

5.2.5 Implementing inheritance via delegation



With delegation the programmer must create the separate objects so that the behaviour and local state of objects is represented separately.

Now an object can be duplicated simply by cloning the 'local state' part of the object, keeping the pointer to 'delegate' back to the behavioural object.

These behavioural objects are referred to as 'traits' object in Self, in JavaScript they are (confusingly as we shall see) referred to as 'prototypes'.

<Diagram source: Antero Taivalsaari, (2009) ref 5.17>

The alert reader may notice the similarities between this and class based languages, explicitly designing abstractions rather than concrete objects is 'not in the spirit of prototype-based programming' [5.18]. There are still differences however, the 'Traits objects' are mutable and new methods or properties can be concatenated, avoiding the extensive refactoring work that the fragile lattice of a class-based hierarchy can create when the system changes.

The disadvantage of delegation is that it requires the programmer to think in an unnatural way about objects, making the programming more difficult, especially for the beginner. The 'class-like' prototype objects are only intended to play the role of shared repositories, but they do not have any particular status in the language. In a complex program they can be wrongly considered as the representation of concrete entities of the application domain.

The advantage of delegation is performance based, to duplicate the Colour Turtle object above the local state object is cloned, with perhaps some of the state values changed. That is all that is necessary. With inheritance via copying when Turtle is cloned, a function pointer to each of the methods is cloned. With delegation only the pointer to the traits object is cloned, no matter how many methods the cloned object has, with inheritance via copying there is a pointer is copied for each method, an object with many methods will therefore take many times longer to clone.

JavaScript strongly favours the delegation approach; it combines a prototype-based object model with a syntax that is recognizable to C++ and Java programmers. This serves to increase its class-like nature, each object in the language has a pointer which it uses to delegate member requests if they are not implemented on the object itself. The programmer can decide what the contents of this object will be, but care must be taken since the delegate object will be shared by all local instances, therefore it makes sense to keep all 'local' state information about the particular instance out of the delegate object.

In JavaScript the delegate object of another object is called its 'prototype' object, and the member look-up chain is known as the 'prototype chain'. The pointer between the two objects is not explicitly available to the programmer but can be set using a special 'construction' procedure involving function objects. JavaScript uses the concept of 'constructor functions' to introduce a more class-like syntax for object instantiation.

5.2.6 Functions as Object constructors

In JavaScript the keyword 'new' is used to make a function object act as an object constructor. For example;

```
function Turtle(){
   // local state variables
   var x = 0;
   var y = 0;
   var heading = 90;
   var colour = 'Black'
}
```

The capital letter in Turtle's signature alerts us that the function is designed to act as an object constructor and that we must use the new keyword to create a new 'instance of turtle'. eg:

```
var sprite = new Turtle ();
```

This line can be more easily understood as:

```
var sprite = new Object (constructor: Turtle, parameters: "")
```

The sequence of this object instantiation is as follows:

1. When the new keyword is used, a new empty object is created, and the Turtle function is called.
2. While the Turtle function is executing the value of 'this' in the function is a pointer to the new object.
3. The Turtle function now sets the properties and methods of the new object.
4. The new keyword returns a pointer to the newly created object.
5. The object will also keep a pointer back to the Turtle function in its 'constructor' property (which can be used to determine the object's 'type' using the instanceof operator).

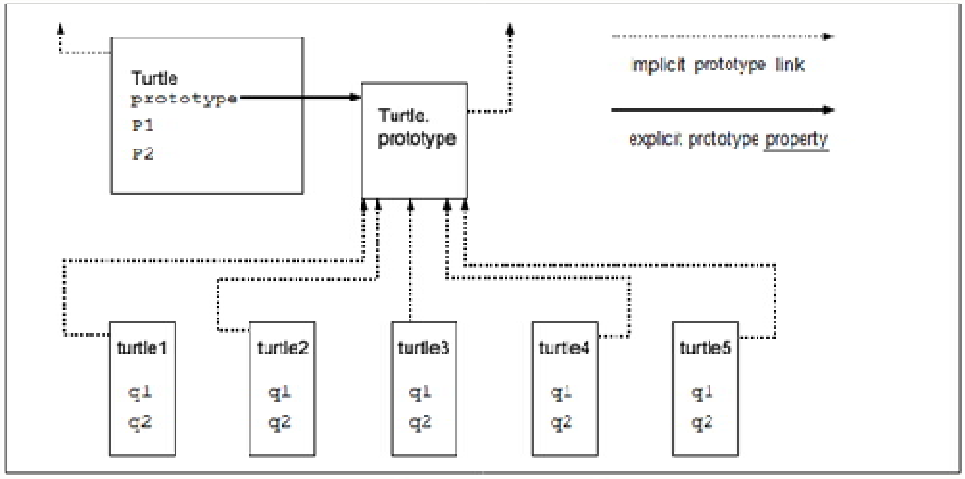
(In a slightly confusing aspect of JavaScript's design, the 'Turtle' function can still be called without the 'new' keyword. In this case 'this' will point to the global object, which is a root container object that represents the global scope, creating properties and methods in that scope. Care must be taken to avoid this.)

The object's prototype chain is initialised during construction. When the 'new' operator is used on a function object (as in our previous Turtle example) each newly created object receives a pointer to the start of its prototype chain. The target of this pointer is determined by the value of one of the constructor Function object's user definable properties, confusingly called its 'prototype property'.

The prototype **object** of the new object is set to the value of the prototype **property** of the constructor function.

This is confusing; care must be taken not to confuse prototype property with prototype object.

Below is a modified diagram from the ECMAscript language specification v5.1 [5.13], ECMA is the body responsible for the specification of the language.



A dashed or solid line denotes the distinction between the prototype (delegate) pointer and the function object's prototype property respectively.

In this diagram the objects turtle1 to turtle5 are created using the 'new' keyword with the Turtle Function object as the constructor. (e.g. turtle5 = new Turtle();) Thus to exploit fully the prototype chain as a method for code-reuse as much shared functionality for the turtle objects must be kept in the object pointed to by Turtle's prototype property.

There is no built in mechanism for creating longer inheritance chains, since there is no implicit access to an objects prototype, it cannot be assigned or reassigned.

It is possible to create sub/super type relationships, such as the one in the turtle/colour turtle example by 'borrowing' Turtle's prototype, attaching it to a temporary empty function object and calling the temporary object with the new operator, (effectively creating a blank object with the prototype link pointing to Turtle's prototype object) This new object can then be set as the prototype property of the constructor function for ColourTurtle – thus an object with a prototype of Turtle will be ColourTurtle's prototype. This is known as constructor chaining. The code looks like this:

```
// -------------- Turtle type definitions ---------- //

function Turtle() {
	/* constructor function to set up turtle instance members */
}

Turtle.prototype.method1 = function(){
	/* start creating prototype turtle shared members (traits) */
}

// -------------- Colour Turtle type definitions ---------- //
function ColourTurtle(){
	/* constructor function set up colourTurtle instance members */
}

TempFunction = function() {} // empty function

TempFunction.prototype = ColourTurtle.prototype() // hijack constructor

ColourTurtle.prototype = new TempFunction();

ColourTurtle.prototype.method1 = function(){
	/* start creating prototype ColourTurtles traits */
}
```

The problem with this is that it requires the programmer to be aware of the details of the prototype-based programming approach. The promise of simplifying the language that prototype system proposed is compromised by the constructor syntax, it exposes low-level details of the language, such as the prototype and constructor properties, directly to the programmer.

In the Smorgasbord framework it would be nice if there were a syntax that more intuitively expressed the prototype/class kludge that is the language's native object model for our developer-user to use, so, using an idea proposed by John Resig [5.14] I have implemented a function object that acts as a 'constructor factory method', the constructor for new sub-types is defined by cloning the constructor of the base type, extending it and automatically setting up the super prototype chain in the process. A new constructor is returned ready to be used with the 'new' keyword to create extended sub-type objects.

The syntax for calling this constructor function looks like this

```
var Turtle = Object.subType();
```

The newly created constructor is extended by concatenating the new functionality - passed to the factory method using an object literal. The syntax for an object literal looks like this:

```
{prop1: value1, prop2: value2}
```

This notation, known as JSON (JavaScript Object Notation) can be used to define both the object an it's methods, for example:

```
{x:0,y:0, forward:function(){ /* statements */ },rotate:function(){/* statements */} }
```

Because JavaScript is almost homoiconic (due to Lisp's influence) this data structure can be in the arranged in the source code to look like a regular piece of JavaScript code - except that the line endings are now a ',' rather than a ';' and where we had '=' we now have ':' e.g.

```
{
   x: 0,
   y:0,

   forward: function() { /* statements */ },
   rotate: function() {/* statements */} }
}
```

Extra care must be taken to remember not to include the unnecessary comma on the last property:value pair.

So the final syntax for the creation of sub-constructors looks like this:

```
var Turtle = Object.subType({
   x:0,
   y:0,

   forward : function() {
         /* statements; */
   },
   rotate : function() {
         /* statements; */
   }
 })
```

```
var ColourTurtle = Turtle.subType({
   setColour : function() {
         /* statements; */
   },
   forward : function() {
         /* statements; */
   }
 })
```

So: we are calling the subtype method with a definition of an object literal in JSON format as the parameter.

To achieve this simple syntax the subType() factory method is initially assigned as a property of the Object method, that is – it is a property of the Object constructor function object (this is very similar to having a static method in Java).

Invoking subType() will create a constructor function with the prototype property set to whatever type of object you have called subType() on, for example if you invoke (a) Object.subType() the returned constructor function will have its prototype set to an instance of Object, if you called (b) Turtle.subType() the returned constructor will have its prototype set to Turtle.

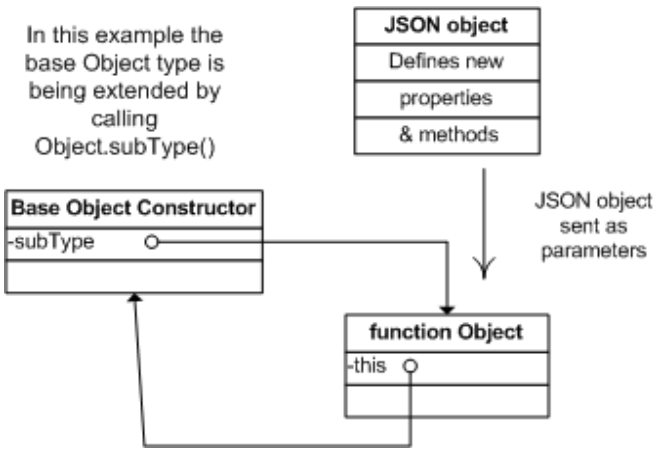
Thus when the newly returned constructor function is used, the prototype chain will point to either (a) an instance of the object type or (b) an instance of the turtle type. The extra properties and methods described in the JSON parameter are copied into the prototype to finally create the extended entity.

Finally in order for the subType() factory method to be available on the new created constructor function (e.g. Turtle) is must assign a pointer to itself to the newly constructed function object.

It is difficult to find a UML diagram to explain this simply, (this would perhaps be a good time to use an animated diagram). I have created a 3 step 'comic strip' of the process to clarify.

Consider that the user-developer wishes to extend the base object type (all sub-type chains must start with the Object type constructor). The base object constructor is a built in language function object named 'Object'.

Step (1)

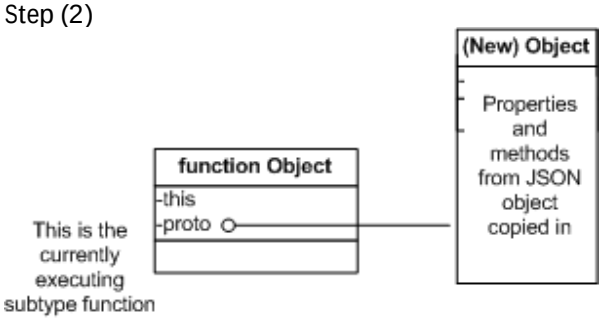


The properties and methods that define how the base type will be extended are sent to the base types 'subtype()' method.

The subtype() method's 'this' pointer points back to the contect from which it was invoked, in this case Object.subtype() was called, so it points to Object.

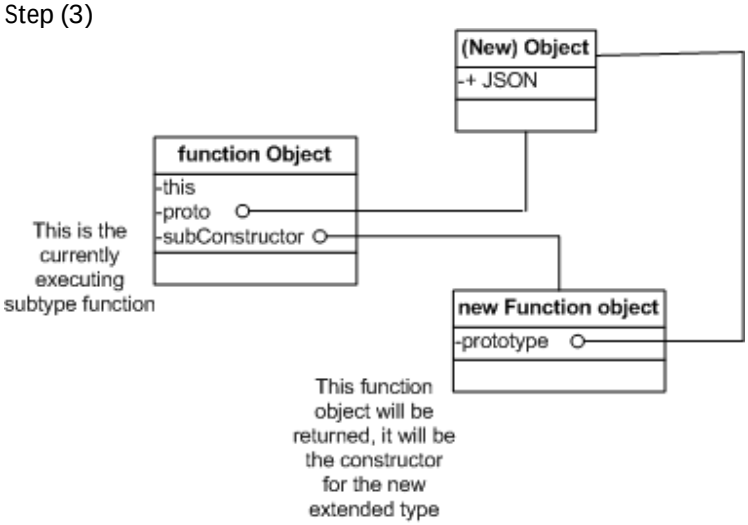
'Object' is now invoked using the 'new' keyword and the returned object is stored in a local variable named 'proto'.

Step (2)



The properties and methods from the JSON object sent as an argument are copied into the new object.

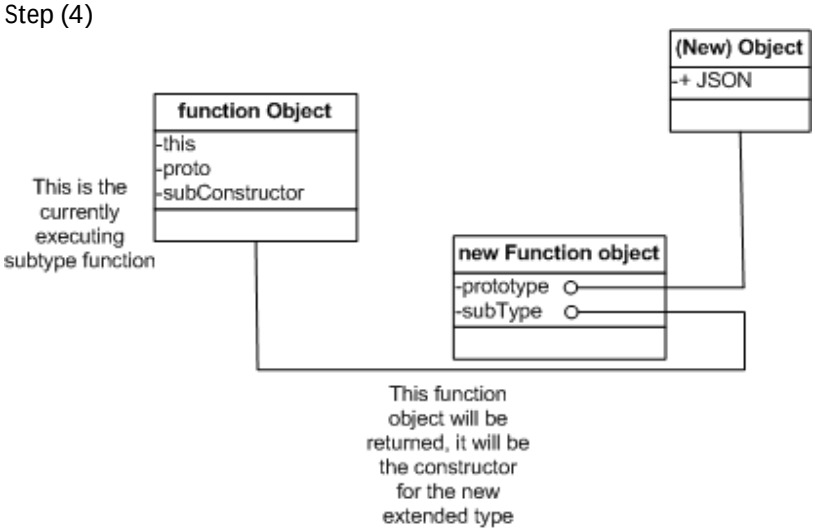
Step (3)



A new empty function object is created and stored with the name 'subConstructor' It's prototype **property** is set to the target of subType's proto pointer.

This function object will be the returned object which will be the constructor function for the new extended type.

Step (4)



Finally a 'subType' property of the new function object is set to a pointer to point back to the currently executing function – which was the subType property of the base Object constructor we just extended.

(We use the javascript arguments.callee property which yields a pointer to the currently executing function)

The new function object is now returned, when it is invoked with the 'new' keyword it will create an object whose prototype chain points to an object of our extended type.

As the newly manufactured constructor function takes a pointer to the subType function with it, it is also now available to be subTyped using the same method.

Much of the power of JavaScript programming lies in this 'conducting a sea of pointers' to various function objects and properties. As well as leading to great power, it can also be very confusing for programmers, hopefully with the power of our 'subType()' method (function object) attached to constructor function objects, we can hide some of this complexity making the developer-users job easier and faster.

N.B. As well as achieving this simplified constructor creation syntax for us, a simple addition allows easy calls to super-type methods that are overridden by the sub-type. The listing for the subType function object is available at: http://catplusplus.org.uk/catpsite/smorgasbord/lib/lang/inheritence.js

5.3 Implementation of the Forces

During the course of the development of the project a new javascript physics simulation library became available. The library, called The Box2d Engine, has been available to C++ programers since 2006. Since then it has been ported to various other platforms including a Adobe Flash version written in ActionScript 3.0. The ActionScript 3.0 version for Flash was close enough to a JavaScript version (ActionScript 3.0 is based on the abandoned JavaScript 2.0 specification, called 'JavaScript with classes') for it to be converted to JavaScript automatically using a transpiler. [5.15]

Having a A-level in maths and not studying physics beyond high school, I defer to the library author, Erin Catto, who holds a PhD in Theoretical and Applied Mechanics in his ability to write a good physics engine.

The library however does not meet our requirements for developer-user productivity, the API is unintuitive and many of the regular routine tasks require a tiring number of lines of 'boilerplate' code. In order to overcome this problem I have implemented an adaptor type called a 'peg' - a peg encapsulates all the functionality needed to set up a body in the box2d engine and connect a smorg's x, y and rotation values to that body's x,y and rotation values in the box2d engine.

Pegs can be subtyped and reused. They can be swapped between Smorgs, so for example if the developer-user has a new smorg with a new sprite in a new game running on new platform with a new graphics rendering pipeline, but the physical behaviour is the same as a previously constructed smorg, the peg can be simple attached to the new smorg. In this way developers who are interested enough can develop peg types and share them with their colleagues and peers. (providing at this point, a 2D bord is being used). The peg types consist of only 2 methods, 'initialisation()' and 'getPeg()'. Below is an example of simple moving box shape which obeys the basic physical laws and responds to collisions.

```
sb.Peg_PhysicsBox = sb.Peg.subType({

  peg:undefined,

  init:function(world,bodyDef,fixDef, owner) {
      bodyDef.type = Box2D.Dynamics.b2Body.b2_dynamicBody;  // the type is dynamic
      fixDef.shape = new Box2D.Collision.Shapes.b2PolygonShape
      if(owner.density){
          fixDef.density = owner.density;
      }
      fixDef.shape.SetAsBox((owner._width / sb.SCALE)/2, (owner._height/sb.SCALE)/2);
      bodyDef.position.x = owner.x/sb.SCALE;
      bodyDef.position.y = owner.y/sb.SCALE;
      bodyDef.angle = owner.rotation * (Math.PI/180);
      bodyDef.userData = owner;
      // now create the object in the physics engine world
      var obj = world.CreateBody(bodyDef).CreateFixture(fixDef);
      this.peg = obj.GetBody()
  }, // init()
```

```
   getPeg:function() {
        return this.peg;
    }
});
```

The method getPeg() returns a pointer into the box2D code, it can be used by the smorg to influence the body depending on user events, for example apply a force due to a keypress, or increase/decrease friction.

5.4 Implementing the Listeners

5.4.1 Keyboard events

At the moment of writing only a generic keyboard listener has been implemented, the factory pattern is used to bind a function to the browser's document object. The document object is at the root of the browsers 'Document Object Model' which serves as the API between the javascript and the browser window.

When the browser window is in focus and a user presses a key, a key down event is thrown, if an element that responds to keypresses, such as a form element, is in focus and that element does not have an onkeydown() method defined (i.e. the onkeydown value of the object is null) the event will traverse the tree from child to parent until it reaches the document element at the root of the tree. The keydown event that is thrown is a simple object, each type of event has its own subtype. One of the properties of the keydown event is the 'which' property - this property holds the keycode of the key that was pressed.

The KeyListenerFactory() function takes 2 objects, keymap and kInfo, these are simple maps. kepmap contains each keycode as a property and a string determining the 'action' we're interested in. keyinfo is the object that the function returns and will track the state of the actions. e.g. in a smorg's initialisation code it may execute lines like this.

```
var keymap = {37: 'left',39: 'right',38: 'up',40: 'down'};
var kInfo = {'left': 0,'right': 0,'up': 0,'down':0}
this.keyinfo = KeyListenerFactory (keymap, kInfo);
```

The smorg will now have a pointer (this.keyinfo) to an object, if for example the left arrow (keycode 37) and up arrow (keycode 40) is pressed the keyinfo object will contain the appropriate boolean values: {'left':1, 'right':0, 'up':1, 'down':0} which can be easily tested using if(this.keyinfo.left){}.

The implementation of KeyListenerFactor is as follows.

```
KeyListenerFactory = function (keymap, kInfo) {
        var key;

        var keyCheck = function (event) {
                key = '' + event.which;
                if (keyMap[key] !== undefined) {
                    kInfo[keyMap[key]] = event.type === 'keydown' ? 1 : 0;
                    return false;
                }
            };

        document.onkeydown = keyCheck;
        document.onkeyup = keyCheck;

        return kInfo;
};
```

5.4.2 Collision Events

For many of the collisions between the smorgs in the system, the developer-user will just want the objects to bounce off each other in the standard Newtonian way, however specific events will be 'special'. Box2d contains the functionality to handle returning data about collisions which have occured but again to keep in line with our aim to increase developer productivity, this needs to be wrapped in a more developer friendly way. Below is the implementation of a 'collision listener' factory which simplifies the set up of collision detection in box2d.

```
ContactListenerFactory = function(callbacks) {

  var listener = new Box2D.Dynamics.b2ContactListener;

  if (callbacks.BeginContact)
  {
    listener.BeginContact = function(contact) // function called on contact event
    {
            // find the info
            var idA = contact.GetFixtureA().GetBody().GetUserData();
            var idB = contact.GetFixtureB().GetBody().GetUserData();
            callbacks.BeginContact(idA,idB);  // then call the callback
     };
   }

   /*
     …… similar methods for other events…
   */

   return listener;
};
```

The ContactListenerFactory() function takes an object which is a simple map defining what should happen when an collision event happens, box2d defines four collision events:

- BeginContact - fired when two fixtures start contacting (aka touching) each other
- EndContact - fired when two fixtures cease contact
- PreSolve - fired before contact is resolved. you have the opportunity to override the contact here.
- PostSolve - fired once the contact is resolved. the event also includes the impulse from the contact.

in the above method ContactListenerFactory first creates the contact listener, then it transfers the 'BeginContact' property (which should be a function object) from the object it was sent as an argument to it's own BeginContact property, which is called when the BeginContact event occurs. The listener function gets the UserData property of each of the two bodies involved in the collision and calls the method from the passed in object with these.

The mysterious UserData property is just a pointer, we set this when we initialise the smorg, it is a pointer back to the smorg which is using the peg.

An example of the ContactListenerFactory() function is used in the demo (demo1) to detect if the cog has hit the spring which returns it to the top of the world.

```
var listener = createContactListener({ BeginContact: function(idA, idB){
    if(idA.name == 'spring'){
       var force = new Box2D.Common.Math.b2Vec2(0, -150);  // up force at 150 newtons.
       idB.peg.ApplyImpulse(force,idB.peg.GetWorldCenter());
     }
}});

this.world.SetContactListener(listener);
```

There is room for improvement here as some of the complexities of the box2d library are exposed, some nice wrapper that allows developer-user to reuse and share code (as in the peg example) is scheduled for future development.

5.5 Implementing Sprites

At the current time of writing only one kind of sprite subtype has been implemented, a simple rotatable one frame bitmap image which I have called the SpriteIcon class.

SpriteIcon is part of the DHTML view system, which uses only HTML elements. This was done in order to maximise compatibility across devices, an alternative view system using the canvas drawing surface and an s.v.g. view system are on the plan for future development.

The SpriteIcon uses <DIV> elements and sets their background-image property using CSS. Briefly for those unfamiliar with browser rendering, there are normally 2 data files:

1. a file of HTML which defines the logical structure of the page, what elements are contained within what, and what type those elements should be and
2. a CSS file which determines how those elements should be rendered.

It is possible to directly set the CSS style properties of a HTML element using JavaScript. However there is a performance issue around that which is due to the way browsers are implemented.

When the browser parses HTML it constructs the Document Object Model, The Document Object Model (DOM) is "a language-independent application interface (API) for working with XML and HTML documents" [5.23] it takes the form of a tree.

In current browsers, since the DOM is language independent, the implementation of JavaScript and the implementation of the DOM are kept separately to each other. In Internet Explorer, for example, the JavaScript implementation lives in a library file called *jscript.dll*, while the DOM implementation lives in another library, *mshtml.dll,* called Trident, Safari uses a library called Webkit originally developed by KDE for DOM and has a separate JavaScript engine SquirrelFish, Similarly Google's Chrome uses Webkit and their JavaScript implementation is the V8 engine mentioned in the introduction, finally Mozilla uses a DOM implementation known as Gecko and a JavaScript implementation known as TraceMonkey.

This separation means that there is a performance penalty for crossing between the two environments, thus as few updates to the DOM must be done per frame as possible, which means update data must be collected together before it is passed to the DOM.

On initialisation the SpriteIcon object requests that the DOM creates a new DIV element, the DOM returns a pointer to the new element which is cached as a property of SpriteIcon object.

When updating, SpriteIcon gets rotation information from the Smorg object that it represents (which comes ultimately from the peg which points into the physics engine) - it does not need to involve the DOM in this, the rotation information is stored in a CSS text string, when the imageDraw method is called in SpriteIcon, location (x,y) information is added to the CSS text string, and this is then passed to the Graphics object. Hence SpriteIcon only updates the DOM on initialisation.

The Graphics object in use for the SpiteIcon is of type DHTMLGraphics - here all the CSS text strings are applied to the DOM, once for each Smorg, and once more to translate/zoom/rotate to the inverse of the ViewerSmorg position/rotation - to give the final rendering the appearance of coming from the ViewerSmorg's position in the world.

Performance has been good enough on a reasonably modern PC (1.66Ghz processor, 1GB Ram, 64Mb video ram) but this could possibly be improved if some way of doing all the updates at once was elucidated.

5.6 Implementing Smorgs

The base type for the smorg defines a relatively simple small object, most of the methods are designed to be overridden and the object is expected to be extended in many ways.

The base smorg code:

```
/**
 * base smorg classs
 */

sb.Smorg = Object.subType({
    // properties (many set as undefined initially)
    OwnerGame:undefined,  // pointer to the game object
    OwnerList:undefined, // pointer to the list of all currently active smorgs
    player:undefined, // boolean true if this smorg is the player
    Sprite:undefined,  // pointer to the sprite object used to represent the smorg
    position:new sb.Vector(0,0), //(x,y)
    x:undefined, y:undefined, //only used in initialising positions
    Name:undefined,


     // methods (many overridden)
    /**
     * takes all the key:value pairs in params object and assigns them to object
     */
    init:function(ownerGame, name, params) {
        //add to smorglist
        this.ownerGame = ownerGame;
        this.ownerList = ownerGame.smorgList;
        this.name = name;
        this.ownerList.add(this);

        for (var name in params) {
            if (params.hasOwnProperty(name)) {
                this[name] = params[name];
            }
        }
    },

    addListener:function() {},

    feellistener:function() {},

    feelForce:function() {},

    animate:function() {
        this.updateAttitude();
        this.sprite.animate(this);
    },

    move:function() {},

    update:function() {
        this.feelForce();
    },

    // setter for the rotation angle
    setAttitude:function() {}
});
```

The most complex type of smorg made so far has been in the demonstration of a multi-part smorg for a realistically steerable car for demo#2 of the 3 online demos (currently firefox only due to vendeor specific CSS property names) at http://catplusplus.org.uk/catpsite/smorgasbord/index.php?demo=2

The construction of a multipart smorg was an experiment, the idea is that the body of the car is not 'pegged' to a physics body but it is attached to 4 wheel smorgs which are pegged to a physics body (each). The peg implemented for the wheel smorgs I called a 'steering peg' and was constructed to have forward motion (relative to its own rotation) but no sideways motion - a more sophisticated steering peg could include some sideways motion at high speed to simulate skidding - each one of the wheel smorgs is placed at the corner of the car body and joined to it. The front wheels are steerable, and rotate in response to the arrow keys (left and right) and the back wheels are powered, which means when you press the up and down arrows you get an application of force, again in the same direction as they are facing.

The car is rotated in the direction of it's motion vector - which means when the front wheels push the car left and the back wheels push the car forward, the car body rotates appropriately as it is 'carried along' by the wheels.

Source code for the carSmorg and wheelSmorg are available online at :
http://catplusplus.org.uk/catpsite/smorgasbord/lib/smorg/carSmorg.js and
http://catplusplus.org.uk/catpsite/smorgasbord/lib/smorg/wheelSmorg.js

In future versions of the software allowing the developer-user to easily create multi-part smorgs through a simple set of commands or interface is planned.

5.7 Implementing Bords

Similar to the base Smorg type, the base Bord is designed in the most part to be extended and overridden. The base does not include any physics, giving the developer-user the option to create without including the physics library.

The source for the base bord type:

```
/**
 * @type sb.Bord
 * Base (bord) class containing state of the world
 */
sb.Bord = Object.subType({

    smorgList:undefined, // list of game objects
    player: undefined,   //  pointer to the player smorg
    border: undefined,   // size of the world
    gameOver: false,     // boolean
    _is3d: undefined,    // used to warn developer-user if they try to use a 2d view system
                         // with a 3d bord or vice-versa

    init:function(params) {
        this.smorgList = new sb.SmorgList(this);   // initialise the 'array' of smorgs
        this.border = new sb.Box(params.size);   // set the limits (in pixels) of the game world
    },

    makeSmorgs:function(smorgs) {
        // put smorgs specified in the attributes sheet into the list
        for (var name in smorgs){
            // the bord picks up a list of smorgs to use in the world from a attribute sheet
            new smorgs[name].type(this,name,smorgs[name].params);
        }
    },
    makePlayer:function(playerProps) {
        var player = new playerProps.type(this,playerProps.name,playerProps.params);
        player.player = true;
    },
```

```
    initialiseView:function(view) {
        // the graphics class is specified in the attribute sheet
        view.setGraphicsClass(sb.attrib.graphics);
    },

    initialiseViewpoint:function(viewPointSmorg) {},  // overridden

    is3d:function() {
        return _is3d;
    },

    // set the size of the world (in pixels)
    setBorder:function(dx, dy, dz) {
      if(dz = undefined) {dz=0.0;}
      this.border = new sb.Box({_lox:0, _hix:dx, _loy:0, _hiy:dy, _loz:0, _hiz:dz});
    },


    // itterate over the smorglist calling the various methods on each smorg
    update:function() {
        this.smorgList.feellistener();// update controller (input)
        this.smorgList.move();
        this.smorgList.update();
        this.smorgList.animate();
        //this.processServiceRequests();   // this is currently unimplemented but design is to
                                           // have a message queue - e.g pause, sound off etc
    }

});
```

The Bord sub-type in demo1 (Firefox only!) http://catplusplus.org.uk/catpsite/smorgasbord/index.php defines a physics simulated world with vertical downwards gravity and a size that is double with height and one and a half times the width of the viewport. Demo 2 defines a physics simulated world with gravity running along a imaginary z-axis (perpendicular to the screen along the line to the user) giving a top-down aspect to the world. Both bord classes override the init() method to do this work. The other methods overriden are those which set the initial position of the viewpoint, and whether the viewpoint must track the player or not (only applicable in worlds bigger than the viewport).

5.8 Implementing View

None of the demos currently need to sub-type the view class. If 3D graphics were to be developed I imagine an extended version of the view class would be necessary. Below is the source code for view, as mentioned in the design section, view delegates every method and as such only exists as a 'template' which is used to drive whatever implementation we have chosen.

```
sb.View = Object.subType({

    _graphics:undefined,// pointer to the graphics which implements the final rendering code
    _viewPointSmorg:undefined, // pointer to the view point smorg which acts as a 'camera'
    _game:undefined, // pointer back to bord class
    _smorgList:undefined, // we cache a pointer to the smorglist locally

    init:function(params) {
        // unpack properties similar to other base classes
    },
```

(script continues on next page)

```
       /**
```

```
        * this method called every frame. It is possible to pass a hint to
        * alter the behaviour of view. The hints are currently implemented as constants
        * globally available (within the framework not within the browser in general)
        */
       onUpdate:function(hint) {
        // check the hint
        if (hint && hint === sb.VIEWHINT_STARTGAME)
        // if the game has just started set up viewpoint
        {
            this._game.initialiseView(this);
            this._game.initialiseViewpoint(this._viewPointSmorg);
        }

        // this is the viewPointSmorgs loop (different to the other smorgs)
        this._viewPointSmorg.feellistener();
        this._viewPointSmorg.move();
        this._viewPointSmorg.update();

        this.onDraw();
    },

    setGraphicsClass:function(graphicsClass) {
        this._graphics = new graphicsClass({_border:this._game.border});
        this._viewPointSmorg.setGraphicsClass(this._graphics);
    },

    onDraw:function() {
        // tell viewpoint to send viewpoint information graphics
        this._viewPointSmorg.loadViewMatrix();
        // get smorgs to execute their draw methods
        this._smorgList.draw(this._graphics);
    }
```

5.9 Implementing Graphics

The base type for the graphics class is very small, consisting mainly of an initialisation function which takes a associative array of key:value pairs and assigns them to object properties of the same name as the keys.

```
/**
 * @type sb.Graphics
 * Base (graphics) class
 */
sb.Graphics = Object.subType({

    _width:undefined,  // the width and height of the drawing surface
    _height:undefined,

    init:function(params) {
        // unpack properties
        for (var name in params) {
            if (params.hasOwnProperty(name)) {
                this[name] = params[name];
            }
        }
    }, // init()

    display:function(hint) {/* overridden */}
});
```

The only currently implemented graphics sub-type is ‘DHTMLGraphics’ which uses HTML elements and CSS strings. Below is a section of the source code of most interest, the bits I have excluded are just initialisation code.

The full script can be seen at http://catplusplus.org.uk/catpsite/smorgasbord/lib/view/graphics/dhtml.js

```
    setPosition:function(x,y) {
        this._positionX = x;
        this._positionY = y;
    },

    /**
     * this method is called by the view point smorg and essentially translates
     * the entire viewpoint appropriate to the its position
     */
    loadMatrix:function(translationVector) {
        var xtrans = translationVector.x + (this._viewportRect._hix/2);
        var ytrans = translationVector.y + (this._viewportRect._hiy/2);

        if(xtrans > 0) {xtrans = 0;}
        if(xtrans < this._limitX) {xtrans = this._limitX;}

        if(ytrans > 0) {ytrans = 0;}
        if(ytrans < this._limitY) {ytrans = this._limitY;}

        elemStyle = this._window.style;
        elemStyle.MozTransform = 'translate('+xtrans+'px, '+ytrans+'px)';
    },

    /**
     * create the world's element in the DOM
     */
    drawbitmap:function(elem, _style) {
        _style += 'top:'+this._positionY+'px;'+'left:'+this._positionX+'px';
        elem.setAttribute("style", _style);
        this._window.appendChild(elem);
    },
```

The important thing to understand with DHTML graphics is that the browser is the rendering engine, and so all we are doing is creating the data for its consumption, in this case it likes its data in the form of CSS text strings, so the job of the DHTMLGraphics object is mainly string construction.

5.10 Other Types

The only other type of note is a vector type, JavaScript has a build in maths library, but it does not include vector maths, needed for much 2D graphics work. Below I include a list of methods this class includes, the full source can be found at: http://catplusplus.org.uk/catpsite/smorgasbord/lib/smorgasbord/vector.js

```
/**
 * base vector classs
 */

sb.Vector = Object.subType({

    x:undefined, // these should be NUMBERS.
    y:undefined,

    init:function(n1,n2) {

    setFromArray:function(arr) {

    setFromVector:function(vec) {

// add this vector to another (returns a new vector this one is unchanged)
    add: function(other) {

// minus this another vector from this (returns a new vector this one is unchanged)
    diff:function(other) {
```

```
// scale this vector by a quantity
    scale: function(by) {

// find the unit vector
    normalize: function() {

// is the magniture almost zero? Useful to optomise game loop
    isPracticallyZero: function() {

//calculate magnitude and return
    magnitude: function() {

    perpendicular: function() {

    inverse:function() {

    /**
     * calculate vector dot product
     * @param {Array} vector [v0, v1]
     * @returns {Number} dot product of v1 and v2
     */
    dot:function(vector) {

     /**
     * rotate vector
     * @param {Number} angle to rotate vector by, radians. can be negative
     * @returns {Array} rotated vector [v0, v1]
     */
    rotate:function(angle){

    normaliseRadians:function(radians){

},

{   // statics
    ZEROVECTOR: [0.0, 0.0],
    XAXIS: [1.0, 0.0],
    YAXIS: [0.0, 1.0],
    ZAXIS: [0.0, 0.0, 1.0],
    PRACTICALLY_ZERO: 0.001
}

);
```

Hopefully, with the comments, it will be obvious what each method achieves. This is also one type which includes Statics.

5.11 Configuring the Framework

I have used JSON as a configuration file, in this class names and paramaters can be stored an a reasonably simple human readable way, that closely resembles CSS. The WC3 is currenly considering 'attribute sheets' which like style sheets will be a standard way of configuring web based applications. I have tried to implement configuration in as close a way to this as possible as I would aim to use attribute sheets to take another step towards giving the develop-user less to learn, or at least if they have not learned attribute sheets, they will gain some transferable knowledge.

The attribute sheet for the demo3 (firefox only!) at http://catplusplus.org.uk/catpsite/smorgasbord/index.php?demo=3 looks like this:

```
sb.attrib = {

game:
{
 bord:sb.Bord2D,
 size:{_lox:0, _hix:600, _loy:0, _hiy:400},
 viewport:{_lox:0, _hix:600, _loy:0, _hiy:400}
},

player:
{
 name:'rmonkey',
 type:sb.MonkeySmorg,
 params:{x:450,y:-250,density:20}
},

smorgs:
{
 lmonkey:{type:sb.MonkeySmorg,params:{x:150,y:280,density:10}},
 floor:{type:sb.BlockSmorg,params:{x:300,y:390, _width:600, _height: 20, rotation:
+0}},
 lwall:{type:sb.BlockSmorg,params:{x:10,y:200, _width:600, _height: 20, rotation:
+90}},
 rwall:{type:sb.BlockSmorg,params:{x:590,y:200, _width:960, _height: 20, rotation:
+90}},
 seesaw:{type:sb.SeeSawSmorg,params:{x:300,y:355, _width:400, _height: 20, rotation:
+0}}
},
graphics: sb.Dhtml};
```

6. Testing

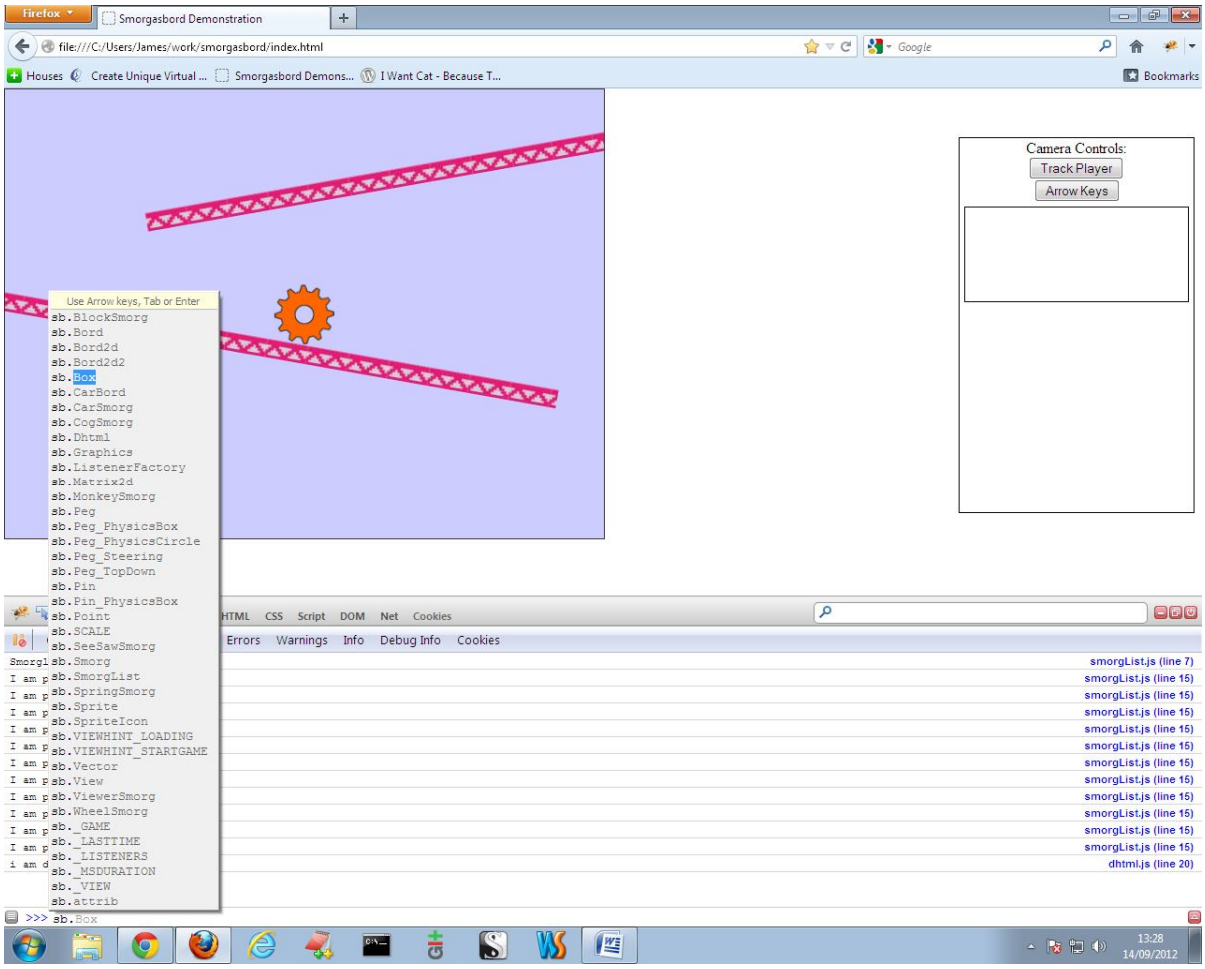
6.1 Unit testing during development

My original plan was to use a test driven environment, by building a testing framework. I thought I would begin development of the test framework after coding a few experiments, that would prove initial ideas worked and initial assumptions were true. However I found to a large extend, the tools I had in place to analyse the results of these initial experiments proved adequate for the rest of the project. I employed the YAGNI philosophy – an acronym coined by Ron Jeffries, one of the 3 founders of the Extreme Programming, which stands for 'you aint gonna need it' – to clarify he wrote "Always implement things when you actually need them, never when you just foresee that you need them."

There are many benefits of test driven development, it certainly contributes much to the final goal of producing error free code, however there is as always a trade-off, and any increase in writing of test code will of course take time to both design and implement. The scale of the project determines the balance of this trade-off – a large project will take a lot of time to comb through using a debugger, hence the time invested up from in writing testing code will pay back with far less or no need to spend time setting breakpoints and looking at stack traces. In a smaller project however, it may be more time consuming to design and implement the test code than the time it saves.

Realistically, this project has produced a working prototype, it is not a piece of software that could be sold or shipped as a product – but the next iteration of the project will take introduce a lot more robustness into the code and this will, as a consequence, I believe require a much more robust development environment which means before I begin to code the next iteration I will produce a test suite.

For this project however, my main testing environment was the REPL style environment provided by the Firefox extension Firebug. The function console.log() is provided to allow string data to be displayed here.



Because JavaScript is JIT compiled on the browser it is possible to have a very quick workflow of edit/test by simply saving the code in the editor, swapping to the browser and hitting refresh, my initial approach was to create mock objects with method stubs – each method would instead of performing any functional work, simply output it's name using the console.log() function. For example, the draw method of the Smorg type would output 'I am the draw method of the smorg' to the console window.

I could have written a test to compare that output string with an expected string, but I found it just as easy to write what I expected on paper, and compare the two lists. This method enabled me to make sure that the basic architecture was in place, no methods were in the wrong scope, the job then is to fill out the method bodies with appropriate functions.

Firebug's debugger has the standard debugging machinery and many times it seemed to me, simple and quick enough to set a break-point on the newly created method body and step through the code line by line, checking the values of variables by tabbing to the 'watch' window where you can see local variables or see a stack trace, where you can see the values of local variables of methods preceding the one you have just coded.

The REPL environment also enables code to be entered and tested live before it is added to the source code, for example it was easy for me to test the results of a chain of vector operations before switching back to the editor where I can add the code I just tested into the correct place in an object method.

This method of working I believe is suitable only for languages that have a very short compile time, so results of changes can be seen almost instantly, running in an environment with a REPL style console and a debugger that features break-points, step into/step over program tracing, a watch window to see the value of all variables and a stack trace which shows all previous methods currently on the execution stack.

It is also only suitable for projects of a certain scale – the Smorgasbord project is around 2000 lines of code, approximately 1,200 lines in the framework and approximately 800 in the three demos. I would say a serious investment in test code begins to make sense for a project beginning at a scale of around 3000 lines and definitely increases its value if there is more than one individual coder working on the framework.

6.2 Testing the framework as a whole

The acid test for the framework was to produce demonstrations of it. Since the overall aim is to increase productivity I gave myself 2 days to create 3 demonstrations. This proved possible – the demonstrations are only rudimentary and can be seen at: http://catplusplus.org.uk/catpsite/smorgasbord/index.php - at the moment this is only available on Firefox – to enable the demos to work on WebKit and later versions of IE is an easy fix (see conclusions section, 'the short view') –

In theory the test would also show that no code had to be altered in the framework, this was however not the case and so this test led to some small amendment to the framework code.

The code in CarSmorg required that the angle of the car wheel be incremented depending on the time passed – since frame updates are not a regular, some way of measuring how much time has elapsed per frame was necessary and the framework did not provide this information. I decided to make this a value available to the whole framework namespace (global to the Smorgasbord but not global to the browser environment) by creating a top-level variable 'sb._MSDURATION' which holds the number of microseconds passed between frame updates, the CarSmorg's code could then include this line to give it the required functionality:

```
var incr=(this.max_steer_angle/200) * sb._MSDURATION;
```

7. Evaluation, Conclusion and Future Developments

Looking back at the original project requirements:

- The developer-user should be able determine the graphics/sound by loading their own media resources
- Support for physics simulation
- Support for some artificial intelligence
- Support for 'character animation'
- Provide Sound/Audio – to provide effects and background music.
- Real-time animation. (the simulated motion per frame is linked to the actual real-world time elapsed between frames). Necessary for convincing physics simulation.
- Animation speed of 50 frames per second (at least 90% of the time).

7.1 Sound/Audio

The first thing immediately noticeable is that there is no sound/audio handling capability in the framework. The audio capabilities of web browsers is much bemoaned by developers of interactive multimedia and games [7.5], nevertheless simply triggering audio files in response to events is simple enough to add to the

framework – the most difficult part of this would be the loading or streaming of audio resources, either sound files will not be available immediately or the system will wait and preload the sounds, the framework should give the developer-user the option to decide with methods in the API for both, for example

```
this.pop = preloadSound('path/to/sound'); //preloadSound a sound object factory method

pop.play(); // use this in response to game events
```

and

```
beginStreamSound('path/to/sound');  // use for long audio files, eg. music
```

7.2 Artificial Intellegence

We have achieved support for physics simulation. I am currently working on some simple AI – for example path-finding and collision avoidance, in a similar way to the keyboard/mouse listeners and 'pegs' into the physics engine, these behaviours should be as much as possible, encapsulated into behaviour methods which are added to any Smorg and invoked when the Smorg is in the update() stage of the main loop.

7.3 Character Animation

Although it is not tested and debugged, another Sprite type – SpriteLoop – which has a changeImage() method has been developed. This sprite uses a technique for fast image swapping commonly refered to as 'css sprites' (7.6) – this technique uses a large background image which includes all the frames of animation you wish to use, often referred to as a 'sprite sheet', and a numerical index determines how far to offset the background image until the correct frame is showing. Thus if the sprite has 10 frames of animation and each sprite image is 20pixels wide, showing frame 6 would mean offsetting the image (translating it to the left) by 100pixels (frame 1 is offset by zero pixels). A sprite could also loop through moving the background image 20 pixels each update, a second row of images with the character facing in the opposite direction could be added and the same 20 pixel per update could be applied although with the vertical position offset by the height of the image.

More complex animation such as inverse kinematics is possible using multiple pegs into the physics engine, and further research into this possibility is planned, one possible barrier to this is that the engine performance will deteriorate the more Smorgs and Pegs are added, a complex animation e.g. a physically modelled skeleton of an animal walking may slow the frame rate to an unacceptable level.

7.4 Real-time animation

As much as possible the animation obeys the principle of real time animation, the framerate does not effect the speed of an Smorg's motion, but instead a slow frame rate makes the motion 'jerky' this most noticeable on slower systems when the garbage collection sub-system of JavaScript is active. Future plans to mitigate the effect of the garbage collector are outlined in the section 'memory allocation' below.

7.5 Frame Rate

The frame-rate of the animations has been surprisingly consistent. I currently have a limited number of platforms to test frame-rate on.

Platform 1: Acer Aspire One laptop with 1.66Ghz, 32bit processor, 1GB Ram, 512 KB L2 cache, 64 MB of video

Platform 2: Toshiba Satellite C660 with 2.53Ghz, 64bit processor, 6GB Ram, 3MB L2 cache, 2GB of video ram

Platform	sb._MSDURATION	Frames per second
Platform 1	35	28.6
Platform 2	17	59

The performance is visually acceptable on the lower spec. platform. Occasionally there are spikes in the value of sb.MSDURATION; this is due to the garbage collection system. I discuss this in the 'memory allocation' section below.

7.6 The Future: Short term

In this section I outline the next one or two iterations of the software, the easy improvements that can be achieved in a short time-scale of a few weeks.

7.6.1 Memory allocation in JavaScript

In lower level languages, such as C++, we can allocate a local object to the stack or the heap. For performance reasons we would like to allocate objects to the stack:

- Stack allocation is very fast as the only overhead is decreasing the stack pointer by the size of the object.
- Stack memory is also very fast as it is 'hot' (used frequently) so it is very likely to be in the cache.

If an object is allocated to the heap, the memory manager will have to find a free chunk of memory and mark it as in use. Heap allocation is therefore slower than stack allocation.

JavaScript is a high-level scripting language. Objects in JavaScript are always allocated on the heap. Since JavaScript was not designed for professional programmers, memory allocation was not designed to be the programmer's responsibility, but instead something that the JavaScript run-time finds the optimal solution for.

There is no way of creating temporary objects in a cheap way and JavaScript run-times are not good at optimizing cases where temporary objects are being used.

In the future on each occasion where a temporary object is used, instead of creating a new local object on the heap, an existing 'temporary' object could be reused. This strategy is known as 'Object Pooling' where temporary objects are fetched from an existing pool rather than created, and returned to this pool rather than being destroyed.

In the next iteration of the framework there will be a strict rule of whenever possible not allocating new local objects in the methods, 'whenever possible' is included in this rule because sometimes we might not have enough objects in our object pool, in this case a method of creating a new object in the pool is provided.

A big benefit to object pooling is it gives us control over garbage collection; since JavaScript is designed to be a simple language, de-allocation of objects from memory is handled by the run-time, using a garbage collector [7.1]. For many applications, this is fine, unfortunately for simulations attempting to simulate 'real time' the running of the garbage collector will cause a noticeable drop in frame rate, the best way to reduce the impact of garbage collection is to reduce object allocation, and have as much control over when object de-allocation occurs. Object pooling will provide this.

The result will be an improvement in performance, the cost of this performance gain is the reduced code readability and increased code size.

7.6.2 Vendor specific CSS

The SpriteIcon module generates data for the browsers rendering engine in the form of text strings. Currently rotation is handled by generating the string:

```
this._style += '-moz-transform:rotate('+smorg.rotation+'rad);';
```

When the browser manufacturers finally get together and commit to the standards process this string will become:

```
this._style += 'transform:rotate('+smorg.rotation+'rad);';
```

It is currently possible to generate the string with both the moz and webkit vendor specific prefixes as the browsers simply ignore each others vendor specific prefixes. It isn't possible to generate the string with both prefixes without performing one more string concatenation operation per smorg object per frame – so it would make more sense to use some method of browser detection and then assign either a function object that generates one vendor prefix or another depending on the browser type detected. This is a very quick fix.

Unfortunately this only fixes the relatively new browser versions. Older versions of Microsoft's Internet Explorer however will need a different strategy since they do not implement the css 'transform' property. They have since version 5 implemented the DXImageTransform property which takes a matrix – below is a rotation matrix.

```
STYLE="position:absolute; filter: progid:
DXImageTransform.Microsoft.Matrix (M11=0.70710678, M12=0.70710678,
M21=-0.70710678, M22=0.70710678)"
```

It is possible to get vector values from the box2d physics engine and since newer browsers can also take the same matix representations as the older Internet Explorer versions, it makes sense for future versions to rewrite the peg and sprite types to pass matrix values to the DHTMLgraphics.

7.6.3 Getters and setters

The final quick fix for the short-term is to implement JavaScript's getters and setters to automatically convert the pixel values used by the majority of the framework to/from the meter values used by the physics engine in a consistent way. Currently a global value sb.SCALE is used to divide or multiply values. This is confusing and should be hidden from the developer-user.

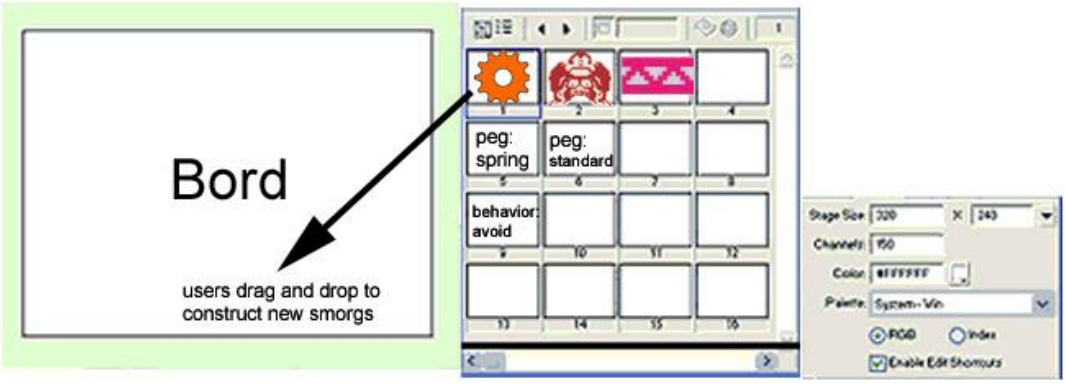
7.7 The Future: Long term

7.7.1 User interface

For the current 3 demonstrations of the software (http://catplusplus.org.uk/catpsite/smorgasbord/ - Firefox only!) hand editing the attribute sheets consisting of JSON code is quick and simple. For larger projects this approach would become unwieldy and frustrating especially to the non professional developer.

A drag-and-drop graphical user interface allowing direct manipulation of objects in a visual way would be an ideal solution to this problem. This would be the first step in creating a web-based application, the developer-user uses the web browser both to create and display their work.

The framework is named 'Smorgasbord' to invoke the idea of a collection of 'parts' that are combined together and presented on the same 'surface'. The interface will carry through that component idea. After choosing a 'surface' – a 'bord' – the user can create a 'smorg' by combine a image or animation, and a set of behaviours, such as how the smorg responds to collisions or gravity, whether the smorg responds to keypresses or mouse events, its initial position and rotation.



This kind of interface suggests that there are two levels of developer-user using the software to create learning material. Users writing code, AI behaviours, physics behaviours and other listeners, and users combining these elements to make the final presentation. Well defined interfaces between components within the framework should enable both reuse and sharing of code. The future of the system then should enable the sharing and reuse of both code and image assets.

7.7.2 Resource sharing community

One of the greatest success stories of the web has been the willingness of developers to share code. There are a great many open source projects and third party 'plug-in' development projects. Some commenters have blogged about the web's 'view source' feature as being an enabling factor in a 'massively parallel development' [7.2]. At this point in time there is an active open source community in the e-learning sector, Moodle, a web based VLE is an open source, community driven project – such is its success that The Open University, the UKs largest provider of over 6000 online courses, uses Moodle [7.3]. There are also numerous extensions to Moodle in the form of plug-ins, the vast majority of which are also open-source community driven projects.

One of the most successful open source projects is Wordpress, originally a blogging platform that grew into a more general website content management system. Current market research indicates Wordpress "Now Powers 22 Percent Of New Active Websites In The U.S." [7.7] – and is the CMS behind award winning high traffic websites such as the Financial Times Alphaville and MTV Europe's Style blog. One of the key factors behind the success of Wordpress is the developer friendly toolset found at www.wordpress.org website, a clear and simple wiki named 'the codex' documents the underlying wordpress framework with examples usage of the API, users can comment and discuss the content of each page using an attached forum. A database of extensions to the frame work 'plug-ins' is available to search, authors of each plug-in can enter a description of their work, a version number, and a link to the documentation and download site. The community can also rate and add comments to the database entries. The database has advanced search and tagging. This feature would is very useful for code-sharing – I would like to develop a similar database to pull

together a community of developers, and a similar community based documentation tool, I hope that developer-users at only a novice level could find and use pre-made components to construct work to fit their learning material.

7.7.3 VLE integration

The construction and dissemination of animations, games and simulations as learning material is only useful to a point, what would be far more useful to educators is if the system could feedback information about what the learner does with the material, how successful the learner was at achieving any goals or challenges the material set for them. If the material is included alongside other material, for example in a module on an online degree, then this feedback ought to contribute to the overall assessment of the learner, this can only be done if the system is integrated into the system being used for assessing the other material. This would generally mean integrating into a virtual learning environment (VLE) – in the future I would like to see the Smogasborg tool packaged as a VLE plug-in, to simplify download/installation and also to integrate the UI into the course administration / course creation interface. The framework, when on the end users machine, could then use events to communicate back to the VLE and which can create usage data helping the admistrators/teachers/trainers of the online course asses the users performance.

7.7.4 From framework to end user development tool

The goal for the framework has always been to increase developer productiveness, each of these long-term goals however points to a simplification of the task, to enable the non-professional programmer to 'program' the final work.

This approach is known as 'End User Development' (EUD) and is a lively research area that combines software engineering and human computer interaction. EUD is a wide subject and there is far too much to cover in this report, a great introduction to the topic can be found at the Interaction Design Foundation website at http://www.interaction-design.org/encyclopedia/end-user_development.html

One sub-component of EUD is End-user programming: A key quote from the above mentioned article is : "End-user programming (EUP) is defined as "programming to achieve the result of a program, rather than the program itself" (Ko et al 2011). In EUP, the developer's goal is to actually use the program; this contrasts with professional programming, where the goal is to create a program for other people to use, often in exchange for monetary compensation." [7.4]

This concept is, I believe, absolutely central to the problem that this project is aiming to solve. Individuals working in educational institutions need to be able to tailor software to their specific needs, they need to be able to be creative with the framework. They don't however need to create a software with the rigour and correctness of a software engineer. They therefore require a different kind of development environment, one focussed on getting the end result quickly, and in short 'getting the job done'. As learning becomes more and more computerised, digital and online, the gap between the skills of the creator of the learning material and the implementer of the material has begun to open up. End User Development provides a direction developers working in the field could use and learn from to fill this gap.

References

[1.1] Mayer, R.E., & Moreno, R. (2002). *Animation as an aid to multimedia learning*. Educational Psychology Review, 14, 87-99

[1.2] *Educational animation*. (2012, July 25). In Wikipedia, The Free Encyclopedia. Retrieved 14:32, September 16, 2012, from http://en.wikipedia.org/w/index.php?title=Educational_animation&oldid=504044703

[1.3] Kenway, J and Bullen, E (2001) *Consuming Children: Education-entertainment-advertising* (Maidenhead: Open University Press)

[1.4] ELDAWY, Mohamed. (2006) *Report on use of middleware in games*. Madison, via <http://en.wikipedia.org/wiki/Game_engine#cite_note-3>

[2.1] White, David and Warren, Nicola and Faughnan, Sean and Manton, Marion (2010), University of Oxford. Department for Continuing Education, Higher Education Funding Council for England, *Study of UK online learning : report to HEFCE by the Department for Continuing Education, University of Oxford.*

[2.2] Belshaw, Doug (2011) Mobile and Wireless Technologies Review. JISC InfoNet <http://mobilereview.jiscpress.org>

[2.3] Richard Noss, Alexandra Poulovassilis, Eirini Geraniou, Sergio Gutierrez-Santos, Celia Hoyles, Ken Kahn, George D. Magoulas, Manolis Mavrikis (August 2012) *The design of a system to support exploratory learning of algebraic generalisation Original Research Article,* Computers & Education, Volume 59, Issue 1, Pages 63-81,

[2.4] Rucker, Rudy V. B. "Projects and Games." *Software Engineering and Computer Games*. Harlow, England: Addison-Wesley, 2003. 15. Print.

[2.5] Gregory, Jason, 2009, "*Game Engine Architecture*". A K Peters

[3.1] McShaffry, Mike. *Game Coding Complete.* Scottsdale, AZ: Paraglyph, 2005. Print.

[3.2] Fine, Richard. "*Enginuity, Part I*." *GameDev.net*. N.p., 28 May 2003. Web. 16 Sept. 2012. <http://www.gamedev.net/page/resources/_/technical/game-programming/enginuity-part-i-r1947>.

[3.3] Rucker, Rudy V. B. "Composition and Delegation." *Software Engineering and Computer Games*. Harlow, England: Addison-Wesley, 2003. 15. Print.

[3.4] *"Model-View-Controller." Microsoft Developer News*. Microsoft, 2012. Web. 16 Sept. 2012. <http://msdn.microsoft.com/en-us/library/ff649643.aspx>.

[3.5] *Unity (game engine)*. (2012, September 11). In Wikipedia, The Free Encyclopedia. Retrieved 15:23, September 16, 2012, from http://en.wikipedia.org/w/index.php?title=Unity_(game_engine)&oldid=511856912

[3.6] "Simple DirectMedia Layer." Simple DirectMedia Layer. N.p., n.d. Web. 16 Sept. 2012. <http://www.libsdl.org/>.

[3.7] *Software development methodology*. (2012, September 6). In Wikipedia, The Free Encyclopedia. Retrieved 15:25, September 16, 2012, from http://en.wikipedia.org/w/index.php?title=Software_development_methodology&oldid=511070244

[3.8] *Smörgåsbord*. (2012, August 30). In Wikipedia, The Free Encyclopedia. Retrieved 15:26, September 16, 2012, from http://en.wikipedia.org/w/index.php?title=Sm%C3%B6rg%C3%A5sbord&oldid=510024876

Material also consulted in this section:

Cwalina, Krzysztof, and Brad Abrams. Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries. Upper Saddle River, NJ: Addison-Wesley, 2006. Print.

[4.0] McConnell, Steve. *Rapid Development: Taming Wild Software Schedules*. Redmond, WA: Microsoft, 1996. Print.

[4.1] Chelimsky, David. The RSpec Book: Behaviour-driven Development with RSpec, Cucumber, and Friends. Raleigh, NC: Pragmatic helf, 2010. Print.

[5.1] Krebs, Brian. *"Java: A Gift to Exploit Pack Makers." Krebs on Security.* N.p., 09 Oct. 2010. Web. 16 Sept. 2012. <http://krebsonsecurity.com/2010/10/java-a-gift-to-exploit-pack-makers/>.

[5.2] *Java performance*. (2012, September 16). In *Wikipedia, The Free Encyclopedia*. Retrieved 15:36, September 16, 2012, from http://en.wikipedia.org/w/index.php?title=Java_performance&oldid=512892987

[5.4] *HotSpot*. (2012, August 5). In *Wikipedia, The Free Encyclopedia*. Retrieved 15:37, September 16, 2012, from http://en.wikipedia.org/w/index.php?title=HotSpot&oldid=505944227

[5.5] Vera, Erick. "Concepts." *ActionScript Virtual Machine 2 (AVM2)*. Adobe Systems Incorporated, 15 Apr. 2010. Web. 16 Sept. 2012. <http://learn.adobe.com/wiki/display/AVM2/1.1 Concepts>.

[5.6] Andreas Gal, Christian W. Probst, and Michael Franz. (2003) *A denial of service attack on the Java bytecode verifier*. Technical Report 03-23, University of California, Irvine, School of Information and Computer Science.

[5.7] Hanselman, Scott. *"JavaScript Is Assembly Language for the Web: Part 2 - Madness or Just Insanity?" Scott Hanselman's Computer Zen*. N.p., 19 July 2011. Web. 16 Sept. 2012. <http://www.hanselman.com/blog/JavaScriptIsAssemblyLanguageForTheWebPart2MadnessOrJustInsanity.aspx> . [Scott Hanselman is chief architect of the Web Platform Team at Microsoft]

[5.8] Garilov, Alexey. "*Bubbles Animation Test: Silverlight (JavaScript and CLR) vs DHTML vs Flash (Flex) vs WPF vs Apollo vs Java (Swing)*". N.p., 13 Mar. 2007. Web. 16 Sept. 2012. <http://bubblemark.com/>.

[5.9] Eich, Brendan. "Every Day I Learn Something New... and Stupid." *Jamie Zawinski's Personal Blog*. N.p., 15 Oct. 2010. Web. 16 Sept. 2012. <http://www.jwz.org/blog/2010/10/every-day-i-learn-something-new-and-stupid/#comment-1021>. [Jamie Zawinski was one of the founders of Netscape and Mozilla.org. This was a comment written by Brendan Eich the creator of JavaScript]

[5.10] *Lars Bak (computer programmer)*. (2012, July 17). In *Wikipedia, The Free Encyclopedia*. Retrieved 15:59, September 16, 2012, from http://en.wikipedia.org/w/index.php?title=Lars_Bak_(computer_programmer)&oldid=502847359

[5.11] *First-class function*. (2012, September 11). In *Wikipedia, The Free Encyclopedia*. Retrieved 16:01, September 16, 2012, from http://en.wikipedia.org/w/index.php?title=First-class_function&oldid=511778855

[5.13] "*ECMAScript Language Specification 5.1 Edition*." ECMA International. N.p., June 2011. Web. 16 Sept. 2012. <http://ecma-international.org/ecma-262/5.1/> [5.15] WebGL Based Game Engine, Martin Nobel pdf.

[5.14] Resig, John (2009). "Object-orientation with Prototypes." *Secrets of the JavaScript Ninja*. Greenwich, CT: Manning. 145. Print.

[5.15] *Box2D*. (2012, August 11). In *Wikipedia, The Free Encyclopedia*. Retrieved 16:11, September 16, 2012, from http://en.wikipedia.org/w/index.php?title=Box2D&oldid=506855733

[5.16] Fossi, Marc. "*Symantec Internet Security Threat Report*." *Symantec*. Symantec, Apr. 2010. Web. 16 Sept. 2012. <http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_exec_summary_internet_security_threat_report_xv_04-2010.en-us.pdf>

[5.17] Antero Taivalsaari, (2009) *Simplifying JavaScript: Concatenation with Prototype-Based Inheritance* . Tampere University of Technology, Department of Software Systems, Report 6, November 2009.

[5.18] Bardou, Daniel. *Inheritance Hierarchy Automatic (Re)organization and Prototype-Based Languages*. Diss. Université Pierre-Mendès-France, 2000 <ftp://ftp.inrialpes.fr/pub/romans/publications/bardou00a.pdf>

[5.19] Henry Lieberman (1986). *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*. In Norman K. Meyrowitz, editor, Proceedings of the 1st Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), pages 214{223, Portland, Oregon, USA, November 1986. Published as ACM SIGPLAN Notices 21(11).

[5.20] Antero Taivalsaari. (1999) *Classes vs. Prototypes: Some Philosophical and Historical Observations. Prototype-Based Object-Oriented Programming: Concepts, Languages and Applications*, chapter 1, pages 3-16. Springer, February 1999.

[5.21] Antero Taivalsaari. 1995. *Delegation versus concatenation or cloning is inheritance too.* SIGPLAN OOPS Mess. 6, 3 (July 1995), 20-49. DOI=10.1145/219260.219264 <http://doi.acm.org/10.1145/219260.219264>

[5.22] Papert, S.A., (1993) *Mindstorms: Children, Computers, And Powerful Ideas*, 9780465046744,, Basic Books

[5.23] Zakas, Nicholas C. *High Performance JavaScript*. Beijing: O'Reilly/Yahoo!, 2010. Print.

[7.1] Mandelin, Dave (2011), "*Know Your Engines - How to Make Your JavaScript Fast*", June 15, 2011, O'Reilly Velocity <http://people.mozilla.com/~dmandelin/KnowYourEngines_Velocity2011.pdf>

[7.2] Shirky, Clay. (1998) *"Clay Shirky's Writings About the Internet." View Source... Lessons from the Web's Massively Parallel Development*. N.p., Apr. 1998. Web. 16 Sept. 2012. <http://www.shirky.com/writings/view_source.html>. [Clay Shirky is the author of two recent books; *Here Comes Everybody: The Power of Organizing Without Organizations* (2008) and *Cognitive Surplus: Creativity and Generosity in a Connected Age* (2010).

[7.3] Dougiamas, Martin (2012). "*The Open University Builds Student Online Environment with Moodle and More*." Online Posting. *Moodle Announcements*. Moodle, 8 Nov. 2005. Web. 16 Sept. 2012. <http://moodle.org/mod/forum/discuss.php?d=34002>.

[7.4] Burnett, Margaret M. and Scaffidi, Christopher (2011). "End-User Development." Encyclopedia of Human-Computer Interaction. Soegaard, Mads and Dam, Rikke Friis (eds.). Aarhus, Denmark: The Interaction Design Foundation, 2011. <http://www.interaction-design.org/encyclopedia/end-user_development.html.>

[7.5] Zugaza, Iker, and Ibon Tolosana.(2012) "Build a Cross-platform HTML5 Game." *.net Magazine*. N.p., 20 Aug. 2012. Web. 16 Sept. 2012. <http://www.netmagazine.com/tutorials/build-cross-platform-html5-game>.

[7.6] Cecco, Raffaele (2011). *Supercharged JavaScript Graphics*. Beijing: O'Reilly, 2011. Print.

[7.7] Rao, Leena (2011). "*WordPress Now Powers 22 Percent Of New Active Websites In The U.S.*" *TechCrunch*. N.p., 19 Aug. 2011. Web. 16 Sept. 2012. <http://techcrunch.com/2011/08/19/wordpress-now-powers-22-percent-of-new-active-websites-in-the-us/>.