Thesis Topic Proposal: Pliant + SDL

Mujtaba Hasni

February 11, 2003

Topic: High-level programming for high-performance multimedia with Pliant + SDL.

Sponsor: Marcus Santos, assistant professor, School of Computer Science Ryerson University.

Abstract

This document is a proposal for conducting experiments with the Pliant compiler and SDL library. It provides a detailed introduction to the two technologies, and Python/Pygame, which is a similar effort for the Python language that this project will draw comparisons to. This proposal then defines some objectives for using the two together. Software and hardware requirements for the project are also specified.

1 Introduction

High-performance multimedia applications, especially games demand direct access to the hardware they run on. Also, these applications tend to be very large and thus also have many bugs.

Games and multimedia applications for the pc platform need to work under many hardware configurations. Because software cannot be optimized for a particular combination of hardware (with exception to game consoles such as those from Nintendo), performance is compromised for compatibility. This means developers need to write to standard APIs such as **SDL** that provide access to accelerated features of video graphic adapters, sound cards and input devices, through drivers supporting the API.

Performance is also compromised for stability and high-level programming. C/C++ is usually the highest level programming language that game developers typically use. Some will use assembler to further optimize code. However, the larger and more low-level software becomes, the more unstable they become. this is largely due to the higher bug-rate. A recent trend is to use C/C++ and assembler to write only the code that needs to execute fast, such as graphics drawing routines, input and audio playback. Games and multimedia use optimized routines for real-time physics, special effects and fast frame-rates. Other elements such as user interface, logic, scripting, file management and data modeling are done in some high-level language. In other words, high-level languages provide a safe, easy way to ”glue” components and optimized routines together.

Python for example uses **Pygame** to link with the routines in the SDL. Higher-level languages are either more expressive and require lesser code to do the same things as lower-level languages do, and/or more simple to use and to understand. High-level languages are usually very safe and may feature facilities such as garbage collection. This usually translates to fewer bugs and faster development time. But most popular high-level languages are interpreted or use a runtime environment, and execute slower than natively compiled objects.

Pliant however, compiles to native code, existing inside memory. Using native code to glue the optimized C/C++ routines may yield performances comparable to software written entirely in C/C++. If so, then Pliant may be a better solution than Python or any other interpreted high-level language for binding with SDL, creating stable high-performance games and multimedia applications with shorter development cycles.

1.1 SDL Overview

SDL, Simple Directmedia Library is an open-source API and library similar to Microsoft's DirectX, and is available on a number of platforms. SDL and DirectX resolves the issue of performance and compatibility by providing software developers and driver writers a standard abstract software layer to allow a program to run reliably and take advantage of the enhanced performance features on infinite number of hardware combinations. SDL comes courtesy of Sam Lantinga. The project is hosted at http://www.libsdl.org.

SDL is written entirely in C and is a self contained library. SDL has facilities for the playback of mpeg, wav, and other media formats[1]. SDL is a popular choice of Linux developers for making media playback viewers. SDL is most popular however for creating games on Linux and MacOS. SDL is a library and an API. If there is a hardware-accelerated feature unavailable or not supported by SDL, it uses built-in, highly-optimized software routines instead. On Win32 systems, the SDL can act as a wrapper to DirextX, providing a comfortable C language layer to an otherwise complicated C++/DCOM/Win32 API. SDL can also safely access the Windows GDI for non-accelerated graphics and standard multimedia interfaces. Currently, SDL is not a good substitute for DirectX on the Windows platform, specifically because it depends on directx for access to accelerated features.

Also because SDL library is written entirely in C (and thus have a simple symbol allocation table), it has many bindings to other languages. Currently, SDL is supported[2] by ADA, C# , Eiffel, Erlang, Euphoria, Guile, Java, Lisp, ML, Pascal, Perl, PHP, Python (Pygame) and Ruby. Notably, Pliant is missing and so making SDL bindings for Pliant is the topic and focus of this proposal.

1.2 Pliant Overview

Pliant uses a dynamic compiler and a trivial high-level meta-language syntax. The Pliant environment offers an HTTP server, FTP server, SMTP/POP3 mail services,

[1] to play media files requires an external codec library of some sort.

[2] language bindings and project urls at http://www.libSDL.org/languages.php

DBS, and the full Pliant OS in 1.4 megabytes compressed, small enough to fit on a $3\frac{1}{2}$ inch floppy disk. The Pliant project is led by its author, Hubert Tonneau.

1.2.1 Pliant 75 Distribution

The current version of the Pliant distribution as of this writing is 75. The Pliant distribution is one of the smallest of its kind available. The distribution provides a suite of programs and low-level libraries called the Pliant Default Extensive Environment, or the PDEE.

The PDEE is designed to make Pliant applications, which run on the internet. For example, the Pliant http server is the official graphical toolkit for Pliant applications.[3] Pliant however does have the beginnings of a graphics library, featuring support for vector graphics – but currently only for X11. The proposed thesis will try to improve this situation by improving performance and portability by using SDL instead.

The Pliant 75 HTTP server can serve regular HTML documents or its own .page document format. A .page document is actually a Pliant program that is compiled and executed when a web-browser requests it. This allows the Pliant HTTP server to serve dynamic content quickly because the processed page resides in memory. Pliant completes the internet suite with an FTP and mail server.

Pliant 75 also comes with the fullPliant OS installation. FullPliant is an operating system built around the Linux kernel and Pliant. Unlike the Linux based systems widely used today, fullPliant is not a unix type OS. The FullPliant OS works as a layer on top of the kernel, providing services which are completely written in Pliant. Then on top of the fullPliant layer is a subset of the debian 2.2 GNU/Linux distribution. This is not included in Pliant 75, but is downloaded off the internet during installation. Debian offers the essential software applications than any user-oriented OS demands, such as web browsers, file managers, etc. This custom debian distribution is a "generic" software distribution that will compile and run on any fullPliant system, regardless of the underlying architecture. There appears however few that are actively using fullPliant.

1.2.2 Pliant Syntax

The Pliant language is designed to be small with new, fundamental features. Even loop constructs such as `while` and `for` are actually routines, written in Pliant. Pliant has facilities for meta programming, allowing extensions to add new features to the language itself.

This is in stark contrast to languages like Perl, which have many features such as file i/o, regular expressions, hashing and list manipulation built into the syntax. The Pliant project intends to implement many features found in other high-level languages to Pliant, though only through sets of external modules.

Like Python, Pliant's syntax is whitespace delimited. Statements are terminated by newlines (or semicolons) and blocks of code are indented. This makes Pliant code appear pleasant in short programs, but more importantly uniform; given a algorithm, any two programmers will produce structurally identical code. However also like Python, long Pliant program code can appear ugly – especially if blocks are nested to deeply.

[3] http://Pliant.cx/Pliantdocs/babel/universal/Pliant/welcome/whatisit.html

1.2.3 Pliant Dynamic Compiler

Using a dynamic compiler, Pliant doesn't generate object or executable files. Instead, Pliant compiles a source file directly into low-level instructions in memory. Pliant can also further compile programs in execution time, similar to using an "eval" statement found in interpreted languages. Pliant supports a form caching by "precompiling" the source code to speed up compilation time for frequently used programs. This is done by dumping parts of the memory core onto file. Like Java's JIT (Just In Time) compiler, code is transformed into native code in one initial process, but unlike Java, Pliant does not make use of a virtual runtime environment. And unlike Python, Pliant is not use an interpreter.

In a traditional static compiler, source is first compiled to assembler instructions. Then the assembler code is assembled into object code. The linker then links all relevant object code to produce an executable file. When this file is executed, the program code is finally loaded into memory and ready to run. Pliant appears to skip all steps between, transforming source to executing program code. Pliant's dynamic compiler goes even further away from this process, allowing parts of the program code to be added, removed or recompiled in memory – without having to reload the modified program core entirely.

1.2.4 Pliant as High-Level Language

Pliant is one of the many high-level languages to emerge in recent years, due in part by the increasing performance of commodity pc hardware. These modern language utilize the processing power to provide facilities such as garbage collection and virtual runtime environments. Similarly, Pliant chooses to utilize extra overhead processing to compile programs at execution time. On current systems, Pliant's fast and lightweight dynamic compiler doesn't present much of a performance hit, especially with precompiling. Once the Pliant program code is in memory, it executes at native compiled speeds.

One of the goals of Pliant and other high-level languages is to reduce the number of bugs and errors in programs. Pliant, along with its standard support libraries can express most programs in fewer statements than C/C++ or assembler can. Smaller code is easier to isolate and debug. Pliant syntax is easier to read and understand; C/C++ can easily lead to obfuscated code, especially if the code makes abuses to the preprocessor.

Another goal of high-level languages is to provide a "safe" environment for programs. Facilities such as garbage collection and exception handling take care of careless mistakes or unexpected errors, without letting programs take the system down.

Pliant is safe to program in, offering rudimentary garbage collection: Pliant uses a simple pointer count system in each allocated object and free's the memory when the count becomes zero. So garbage is taken care of as long as memory pointers don't indirectly point to them selves. This memory management model falls somewhere between low-level (C/C++) and high-level (Java, Eiffel) memory management.

Hubert Tonneau believes that current high-level memory management is inefficient for large data allocations, so Pliant's choice is actually a compromise. Still, because of the modularity of Pliant, any kind of memory management model can be implemented,

and appear as an extension to the Pliant language itself.

Pliant is a language that can express things at high level or even at a low level. The Pliant default extensive environment (PDEE) has useful, reusable and extensible routines to make programs at a very high-level. And Pliant can also go quite low-level, offering in-line assembler[4] when required.

1.2.5 Pliant+SDL Considerations

Pliant has facilities for directly linking with dynamically linked libraries written in C. SDL is written in C, so writing bindings for each of the SDL routines in Pliant is a no-brainer. If the project would stop there, then the result would be an API identical to the C based API of the SDL. But more work will be needed to make full use of the benefits in the Pliant language.

When making the Pliant+SDL bindings, its important that it is designed so that littlest possible coding is necessary to use SDL. This project could simply implement the bindings to match exactly the C API of SDL. This however would mean that the Pliant code would resemble its equivalent in C, offering few real advantages over using C itself. The goal then should be to make using SDL easier through Pliant. Python's Pygame module may offer clues on how this can be done. Much of the project's time will probably spent on this problem.

Pliant's high execution speed and simple language could make it very useful for authoring multimedia applications and games. By linking Pliant with SDL, Pliant should perform at speeds comparable to C with SDL – more so than Python with Pygame. Also, the amount of coding required in Pliant can be significantly less than in C and about equivalent to Python. It depends on the design of the Pliant+SDL bindings that thesis wishes to develop.

1.3 Python Overview

Python is high-level, object-oriented programming language that is safe and easy to use. Python easy to embed in programs, and is becoming the language of choice for "scripting" or automating software applications. This makes Python a scripting language. Python is also extensible using C or C++. Python is named after the bbc sketch comedy series, "Monty Python's Flying Circus." Python is written by Guido Van Rossum and the project is hosted on `http://www.Python.org`.

1.3.1 Python 2.1 Distribution

The Python2.1 distribution appears to offer a complete suite of tools to create cross-platform software applications of any type. The distribution offers tools to write, design, compile, debug and document Python programs and offers a rich library of reusable classes. These include classes for making cgi scripts, server pages, system reporting, database access and applications using graphics and GUIs.

[4] available through asm module

Python 2.1 comes with packages to access any widely used system or libraries – more than Pliant[5] currently has. These include packages such as Tkinter for making GUI applications. Python 2.1 also come with helpful support programs such as IDLE, a complete Python editor and integrated development environment that is written in Python itself.

Pydoc scans Python sources and creates documentation in html format. Pydoc can also function as an HTTP server, serving up documentation of the Python source tree in real-time. Pydoc by default works like a man command for Python classes.

Like Pliant, Python also has classes for creating an HTTP server, but also with support for CGI. Pydoc uses this class. Unlike Pliant however, Python doesn't provide a complete suite of internet services beyond HTTP and the server socket.

Another interesting packages include "freeze", for making stand-alone Python programs without the need of the Python distribution itself. Freeze allows Python programmers to create stand-alone programs by cloning the Python interpreter, and appending it with a compressed archive of the byte-compiled program source the Python classes it imports.

Freezing a Python program means the end-users do not have to have install a Python distribution. This process however produces a single over-sized executable, which doesn't execute much faster than under the original Python environment. Though Pliant uses a compiler, its design intentionally avoids compiling to an executable file.

Neither languages are suited for creating and distributing stand-alone applications like C/C++ traditionally does. This "feature" that both these language share discourage commercial developers from exploring the possible applications of these languages for video-game and multimedia software.

1.3.2 Python Syntax

The Python syntax is white-space delimited, and as a result the structure appears somewhat similar to Pliant. This is where the similarity ends. Python has many built-in constructs and functions. Pliant has a very basic few built-in constructs, but has more expression operators. But because of Pliant's extensible meta-language, it can make itself appear even more similar to Python if desired.

Python is extensible, but only through C or C++. Pliant can extend itself using its own language. Python requires special modules to gain low-level access. For example, Python requires modules to perform conversions between packed c structures, to Python strings, then back again. Pliant natively supports packed data structures and even C prototypes. And unlike Python, Pliant can also program at the low-level as well. This is a typical differentiation between interpreted and compiled langauges.

1.3.3 Python Interpreter

Python is an interpreted language. This means that programs are not executed as machine code and the source is a script for the interpreter program. Python is also known as a scripting language. Python however is quite fast, especially on current commodity

[5]note: library bindings packages are quite trivial to write in Pliant, in comparison to Python. Also, the Pliant distribution tries to maintain its svelte, floppy-disk size

hardware. Python can compile Python source to byte-code, which load much faster than the raw source. Byte-code however is still dependant on the interpreter and is far from being native code that Pliant can generate.

The main advantage of using an interpreter is that the language is dynamic. This allows for dynamic data typing and data types. Python is also interactive so users can enter Python code directly into the interpreter. For many programmers, using an interactive mode is an easy way to learn and experiment with a new language. also, Python programs can also further evaluate Python code. Pliant can at least perform the latter, because of its dynamic compiler.

1.3.4 Python as a High-level Language

Python is highly successful because of its powerful, but easy to use and understand language. Python is object-oriented, unlike Pliant. Pliant does implement some features such as virtual or generic data types, but does not implement inheritance, which Hubert Tonneau feels lead to unnecessary bugs[6].

Beginners and novice programmers find Python very easy to learn. Python's syntax is quite clean and easy to understand. Python provides a safe environment for programmers to work in, offering full garbage collection and exception handling. This means that programmers do not need to be as careful about low-level matters such as memory allocation, null pointers and data types as they would if they were programming in C. Python works the way most people who understand the language expect it to work.

Python is ideally suited as a true scripting language, being used to glue components together, and provide overall programming. Because of Python's object oriented nature, it works well with components written in C++ or other object oriented languages. Python can also interestingly take components written in c or other procedural language and apply them in an object oriented frame-work. This is what **Pygame** essentially does for SDL.

Python is also popular for scripting on the web. Products such as zope provide a frame-work for creating interactive web applications, portal sites and group-ware. Cheetah, is an extension to allow Python to be used in "server pages" much like the way PHP is being used.

Another important and popular use for Python is as a RAD (rapid application development) tool. Using Tkinter or wxPython, Python can be used to easily and quickly build useful full-fledged applications, featuring GUI's, networking and database access for example. This allows developers to prototype software ideas in shorter time cycles. Python is also ready for making enterprise[7] software as well. Python may very well soon compete with Visual Basic.

1.3.5 Pygame: Python + SDL

Pygame is a Python module providing Python access to the SDL API/library. The project is hosted on `http://Pygame.org` and is currently maintained by Pete Shinners.

[6] http://Pliant.cx/Pliantdocs/babel/universal/Pliant/welcome/whatisit.html
[7] check http://www.activestate.com

Pygame makes game creation and multimedia application development easy. Pygame works at even higher level than SDL does, supporting a subset of routines in SDL and simplifying the rest. Pygame is almost a separate API exclusively for Python. Python and C are very different langauges. The SDL API was designed for C programmers in mind (though choosing C over C++ implied a desire to have bindings to other languages as well). Pygame's API is adjusted so that it works well within the Python style. This is really how Pygame makes development easy, by not trying to copy the C API exactly, yet begin still familar to other SDL users.

Unfortunately, Python cannot program at the low-level and so cannot use any low-level code beyond that Pygame makes available. The Pygame module is written in C, in the Python API. The Python interpreter links with dynamically linked libraries through its C interface. Pliant on the other hand directly supports low-level C libraries. This means that the Pliant module for SDL, the basis of this proposed thesis, can be written in Pliant's own language. Its possible for Pliant to support every low-level routine in the SDL, and can even use low-level code outside the SDL, for example opengl. Pygame currently doesn't expose bindings to opengl extensions in SDL.

1.4 Cross Platform Development

Cross platfrom development means how well programs can be made to run on different platforms and architecutres. C has been and still is the most portable language used in the industry.

The SDL is written in C and has been ported for all architectures running Linux, Windows, mac os, beos, freeBSD, unix and even for console machines such as Playstation 2 and soon to support modified x-box units. This means that any game or multimedia program that exclusively uses the SDL library can compile and run on any platform that SDL has been ported for. This makes SDL the best cross-platform development tool for making high-performance multimedia and games.

Recent years however has shown a preference for more specialized, higher-level programming languages. Much of these "modern" programming languages try to make them selves independent of the underlaying operating system and architecture. Languages such as Java use a virtual runtime environment, perl and Python use an interpreter to read programs in the form of text or bytecompiled files. Pliant uses a dynamic compiler, written in C that compiles from source .pli files into program code native to the host system.

All of these language have interfaces that allow them to link with code native on the host system. Java uses jni, wich links to native code through the jvm, perl and Python use extension modules written in c, but in the API compatible to these languages. Pliant programs executes as native code, so it easily links directly with libraries, which are compiled in native code.

currently Python is available on almost all systems that SDL has been ported on. This is because Python to is written in C. This works favorably for Pygame, which fully supports Linux and Windows, as well as other systems that can host both SDL and Python. Work on mac os and osx versions are currently underway.

Pliant is available on Windows and posix compliant systems. Pliant is written in c and a subset of posix. Pliant requires a multi-threading environment with TCP/IP suite.

This is due to Pliant's current distribution which employs a number of internet services such as the http. This however doesn't affect Pliant's portability because most systems, even Windows support a basic subset of posix, providing at least TCP/IP networking. Pliant should have little problem working on all platforms it supports with SDL.

perhaps this may suggest an even further trimmed-down Pliant distrubition designed for SDL? it would provide an even more cross platform development tool, opening possibilites on platforms that do not have tcp/ip or even an os, like game consoles.

2 Objectives

the overall objective is to produce a tool for creating high-performance videogames and multimedia applications that is easy to learn and understand, and require as little coding as possible. Pygame reasonably meets this objective using Python, but lacks the performance of C/C++. It is hoped that making a similar package for Pliant will offer the simplicity and ease of use that Pygame offers, and also the performance of C/C++ and offer possible extensions for 3d graphics with opengl. Along the way, the project will be documented on a website, hosted by the Pliant http server and using Pliant's pages format. But before any of those objectives can be met, some aspects of the aformentioned technologies will need to be studied and learned.

2.1 Learning Objectives

the proposed thesis will be undertaken with minimal knowledge and experience with almost all involved technologies. This may mean that much of the project's life cycle will be spent learning the languages and APIs, and also how they interact and connect with external elements. Not all learning objectives need to be met first, but rather met along the way through the project cycle. Indeed, the thesis process itself is a learning process; new learning objectives can be added or even replaced. The following defines some of the things that may need to be learned.

2.1.1 Pliant

the Pliant language is small, but unique. Learning to write in the Pliant language can be picked up quite quickly. But learning to take full advantage of Pliant's features may take considerable time. For example, its meta-programming facilites demands extra attention. There is also the question of how well Pliant can use SDL, which is a C library.

2.2 Pliant and C Libraries

how well Pliant links to C compiled libraries will be crucial to the success of Pliant+SDL. On the surface, it appears that Pliant can directly load dynamically loaded libraries, or DLLs. But certain questions like how does Pliant handles library dependancies? for example if SDL depends on `libfoo.so.1`, does Pliant know this and load `libfoo.so.1` or does the calling c-library take care of it? and does this work on

both the Windows and Linux platform? more knowledge about *how* Pliant uses DLLs is needed.

another questions is how Pliant deals with packed data in c-structs. The SDL library makes heavy use of structs to pass information through its routines. Pliant supports data structures and user type definitions. Again, on the surface it appears that Pliant's data structures directly support C, but further study is required.

because SDL takes control of hardware directly (through available APIs), it must be certain that Pliant will not unpredictably get in the way of SDL's normal operation and interaction with the underlaying hardware. Again, understanding what environment Pliant loads DLLs and execute the contained code in is required here. Hopefully, it will be found that Pliant programs will surrender appropriate control to SDL when its used. Otherwise, the bindings may become more elaborate than simple function prototypes.

there are some examples of binding modules to actual c-libraries in the Pliant 75 distribution. These files need to be examined and some test cases need to be conducted.

2.3 Pliant Modules

learning how to write Pliant modules should be easy. What needs to be learned is how to make modules that are well designed for the Pliant language. This means that the Pliant+SDL module must offer a wrapper API that fits well in the Pliant language. Almost all the support files making up the PDEE are Pliant modules, so there are pleanty of examples to follow. Notably, the graphics library contains interfaces with Xlib. This package may offer clues on how to convert a C-style API into a useful Pliant-style API.

2.4 Meta Programming

meta-programming is a powerful tool that Pliant uses to extend its self. Using this, Pliant can extend its vocabulary and syntax, sometimes to the extent of making itself appear as a new language. For example, the .page format that the Pliant http server uses are actually Pliant programs even though they appears as if written in a different language. Pliant offers the kind of power and flexiblity usually found in functional-programming langauges such as lisp and scheme, for those who know how to use it.

2.5 Embedding Pliant in C apps

still today, industrial strength applications are written in C and C++. Pliant tries to replace c, but the reality makes apparent that Pliant will have to find a way to work inside C. What needs to be learned is how to integrate or "embed" Pliant in a C program.

this would make Pliant+SDL useful for current game/multimedia developers. They can create a high-performance frame-work in C/C++ and then use Pliant as a high-level scripting language to simplify creation within the frame-work. For example a game-engine, written in C/C++, could use Pliant as the language used to describe game objects, levels, rules, menus, etc. Plugins and mods could also be written in Pliant.

Crystal Space[8], a powerfull game engine for creating games with 3D graphics already uses Python as its scripting language.

the objective here is to allow C programs to use Pliant objects by using an embedded Pliant dynamic compiler. Pliant is written entirely in c, so it is very possible to do this. How well Pliant easily lends itself to doing this needs further study.

2.6 Python Language

the Python language is an easy high-level language. Its structure and design is simple to follow. Python 2.1 has a rich library of classes, that makes up most of Python's functionality. The objective is to learn how to use both the Python class library and its built-in features together.

2.7 Python Binding to C

a learning objective is to better understand how Python uses C modules. Python bindings come in the form of modules which provide Python with usable classes. but any classes that these bindings offer cannot be subclassed. That's because the bindings are compiled in c, using Python's API. This proposed thesis will examine and compare, how well Pliant binds to C in comparison to Python. This will include sample code from both languages.

while Python may already not offer the most optimized solution for scripting C/C++ programs, there are good reasons why its very popular for doing this. A major factor is Python's friendly approach to object-oriented programming. Python could in a way be a tool to easily create object-oriented applications using standard C libraries. Python has C++ wrappers for its C API. This means that Python can also pass and receive messages to and from C++ objects.

Pliant doesn't offer full object-oriented support, especially lacking class inheritance. Its still theoretically possible to extend Pliant to offer full object-oriented support, but the Pliant team doesn't appear interested in going this direction. They feel that it leads to unnecessarily complex projects, which in turn leads to bugs. some would argue the inverse. What this all means is that for the majority of game programmers who perfer object-oriented programming will likely prefer Pygame over Pliant+SDL even if it performs better.

2.8 SDL & Pygame

The SDL is a simple API as its name says. To learn the API and how to use it, some experiments will be done in its native C language. The SDL team doesn't pledge official support for any other languages other than c, though it encourages efforts to offer bindings to other languages. Learning the API, its important routines and data structures is important. Pliant needs to know the names of the routines it will call directly in the library. And all of the c data structures that each routine uses and returns may need to be ported into Pliant data structures.

[8]More information at `http://crystal.sourceforge.net`

Hopefully, learning all of the SDL won't be necessary. Like Pygame, only a subset of the higher-level routines in SDL will need to be called directly. For the proposed thesis, the bare minimum of SDL required to write a game and a movie player will be "exposed" to Pliant. From there, the thesis will try to match as many of the features that Pygame exposes in SDL.

Also, Pygame will be learned. Pygame is in someways its own API wrapped around the SDL. Pygame is designed to be even simpler than SDL and made exclusivley for Python. Understanding the differences between the C API and Pygames will be important when making decisions on how to adapt the C API for Pliant.

2.9 Cross Platform Support

Pygame supports development on Windows systems as well as Linux, as well as many other. Pliant+SDL should attempt to support at least Windows and Linux. What needs to be learned are the differences between Pliant on Windows and on Linux. Can Pliant load DLLs as well as it can in Linux? there are differences between Windows DLLs and Linux shared objects that may need to be understood.

Pliant is compiled under cygwin, a posix emulation layer ontop of the Win32 API. It may be required then to compile SDL under the same environment. But if Pliant can be compiled with native tools, such as mingw or visual C++, then maybe plian can load Windows standard DLLs. One way to be sure is to download the Windows build of Pliant from the project site and test it. It it doesn't work, then the next step is to figure how to compile Pliant without cygwin. The last thing to try if all else fails is to compile SDL under cygwin. There might however a performance hit on SDL it it were forced to use an emulation layer to work in Windows, but its uncertain now.

3 Project Objectives

The project's objectives are outined below. Time is short, so some maybe even most of these objectives may not be met. The project's main objective howerver is to provide the tools and framework to make all of these objectives implementable given enough time.

For the objectives below which actually use the Pliant+SDL bindings, they will be replicated in both Pygame, and standard C + SDL. Then simple comparisons will be made. Each of the three versions will recieve a rating based on performance and program code size/readability. It hoped that Pliant will have the highest ratings in most if not all objectives.

3.1 Project Website

The project life-cycle, begins after this proposal and will be tracked on a project website at `http://playground.scs.ryerson.ca:9090`. the website is hosted on a Pliant http server at port 9090. The website will be written in Pliant's .page format. While this aspect of Pliant is not officially being explored in this proposed thesis, the .page format provide useful facilites for exhibiting Pliant projects. These include

side boxes, syntax highlighting and facilites for browsing deploying Pliant software projects.

The website will have an open journal, covering the development of Pliant+SDL, lessons learned, and e-mail correspondance with ryerson staff, other experienced Pliant, Python, and SDL users, and project members and assitants. All deliverables, including the official thesis paper will be made available on the website. The look and feel of the site will be consistant with other Pliant sites, though slight changes such as color may happen if the project takes off and define a style of its own. This is typical of game development sites, for example the Pygame site.

The objective of the site is to be a base for all work on Pliant+SDL. The site will be a source for all material related to the project, including source, binary packages, documentation, thesis works and links to all other related material used from other projects. The website will have sections dedicated for some of the objectives defined below. Each section will track the progress of each objective. There is no set order or time-table, beyond the overall deadline, in which the objectives need to be achieved. Some of the objectives will be approached in an interspersed manner, the focus of work may change as new things are being learned and as obstacles appear. The website will help keep track of all these changes to make the whole process safe and painless.

Once the Pliant+SDL bindings prove usable, the website will host simple tutorials on how to use them together. Hopefully, it may invite others to try and test Pliant+SDL and perhaps support it, even after the thesis is concluded.

3.2 Pliant+SDL Bindings

The objective of the Pliant+SDL bindings is to either offer Pliant modules that help Pliant interact with the SDL library, or offer some instruction or documentation on how this can be done. Pliant may not require any special module to link with SDL, but it may need some headers[9] to define SDL data structures and function prototypes. This objective is the bare minimum that this thesis wishes achieve.

Pliant+SDL bindings will try to expose SDL routines to support:

- Windowing & fullscreen graphics drawing context/canvas.
- Keyboard & mouse i/o.
- SDL event handling.
- Complete 2d drawing routines.
- Threading support.
- Timer routines.
- Digital audio.
- Bitmap & movie[10] formats.

[9] "headers" is a C term for a file contiaining pre-declared symbols, its understood that Pliant does this differently.

[10] movie playback is not supplied by SDL.

3.3 Portability

Portablity is more important now because of the many available platforms and architecures that are being widely used. There are personal computers such as the pc clone and apple macintosh (amiga[11]is hoping to make a comeback soon). And there are game consoles such as sony Playstation 2, nintendo game cube and microsoft x-box. Game developers often release a game accross manny of these platforms, so a cross-platform frame work becomes useful.

like Pygame, Pliant is available on Linux and Windows. Combined with SDL, they may offer a solution for creating prototypes or re-usable modules that can work on many platforms with little or no modification. Pygame already has success, supporting Windows, Linux, *BSD, MacOS x and even Playstation 2. This however is due more to the vital support Pygame has than the porability of Python and SDL.

The objective is to examine how well Pliant+SDL programs work on both Linux and Windows. Hopefully, the binary distributions for Pliant and SDL for Windows will suffice since all new code will be written in Pliant. If however recompiling or modfication of either's sources becomes necessary, a Windows C/C++ development package may be required. Because Pliant and SDL are free and open-source, this part of the project will try not depend on commercial, propriatary products such as Microsoft Visual C++. If no free tools such as Cygwin gcc or Mingw gcc offer a solution, then Pliant+SDL will exclusively be developed for Linux.

3.4 Special FX Demos

Some special effects require computational power, aside from that required to display graphics and generate sounds. These include particle effects, real-time physics and morphic geometric transformations. These are examples of code that typically doesn't reside in standard APIs such as SDL or OpenGL.

Performance is where Pygame by design fails. The performance of these effects depend on the execution speed of the Python interpreter. This means that complex special effects will have slower frame-rates leading to unconsistant display and rendering. The more work Python needs to do, the slower Pygame performs. Yet Pygame is an ideal place to design special effects because of the simplicity of the Python language. Pygame could be used as a prototyping tool for exploring new ideas for special effects, before being ported into optimized C code.

Pliant+SDL aims to offer the simplicity of the Python language, with the perfomance of C by nature of its dynamic compiler. To test this, a single special effects demo will be ported onto C and SDL, Pygame and Pliant+SDL. Then some performance benchmarks and eye-witness accounts will be recorded. Which special effect to be used is yet unspecified. It however will ideally be simple to implement, yet intensive computationally.

The objective is to prove that Pliant can perform a common special effect with the performance comparable to C, and code size and simplicity comparable to Pygame.

[11] amiga is still alive, `http://www.amiga.com`

3.5 Pliant+SDL API

To make Pliant work with SDL doesn't actually require any special binding module like Pygame for Python. Pliant can make calls directly with libsdl, given the approriate prototypes and type definitions. So the "bindings" will simply be a file listing header information that can be included in a Pliant+SDL program.

The Pliant+SDL API will be a wrapper module designed to make using Pliant with SDL easier. The objective is to provide an easy path to using SDL routines which doesn't require declaring prototypes directly in the program.

The first and most basic version of the Pliant+SDL API will simply mimick the C SDL API, requiring all components to be initialized and registered. This would mean that Pliant+SDL code will not be significantly smaller than if done in C.

The second version and final will look towards Pygame's way of wrapping the SDL. Pygame is object-oriented, so some adjustments will be made. The goal is to cut coding size and ease to being equivalent to Pygame or better.

A third, experimental version will use Pliant's meta-programming facilities. This is a tertiary objective, to provide an even simpler way of using the SDL by wrapping the SDL in an extended version of the Pliant language. The .page format is actually a Pliant program, but it appears different then conventional PDEE programs. This is because Pliant allows programmers to customize the language to add new keywords and constructs. The purpose is to examine what role Pilant's meta-programming features can play in Pliant+SDL.

3.6 Interactive Game $<$ 500 Lines

The game itself is left unspecified. It will be a simple game with simple gameplay. This game could be as simple as "punch the monkey" as seen on banner ads. The 500 lines is an arbitary number, which is small enough for one to study in one reading. The point is to show that a game is possible under 500 lines with Pliant+SDL.

Some game ideas to be considered are (in order of complexity):

Punch Monkey
: Simple punch the monkey game.

Darts
: A level up from Punch Monkey. Player throws darts on a board. Features a "nervous"(uncontrollable, spuratic movement) targetting system.

Astroids
: Arcade classic, player's space ship shoots astroids that break into smaller ones. Features effects of gravity and interia. Top view, fixed playing area.

Outer Ridge
: Player destroys incoming astroids. Target/Cockpit view, real-time bitmap scaling astroids become bigger as they approach. Star field effect back-drop. Features scrolling game field simulating a ship rotating 720 $^\circ$.

3.7 Movie Player

The movie player will be a simple mpeg viewer written in Pliant. SDL offers the facilites for displaying moving graphics in a window, the ability to generate digitally synthesized sounds using a sound card, and the ability to handle user interaction by keyboard and mouse. There are many mpeg viewers such as gtv that use the SDL. But SDL has no built-in codec for reading mpeg or any other movie file. Since writing an mpeg library in Pliant is outside the scope of the proposed thesis, an existing mpeg library will need to be used.

This means that yet another set of bindings for Pliant will need to be made. Depending on how difficult writting the SDL+Pliant bindings become, an attempt will be made to provide bindings to the SMPEG library. SMPEG is an open-source project provided by Loki software, the once preimier but now defunct developer and importer for PC games on the Linux platform. The library is actually the blending of UC Berkeley's mpeg_play mpeg video decoder and the SPLAY mpeg audio decoder libraries. There are many other quality, open-source mpeg decoder libraries available such as mpeglib and libmpeg2, but SMPEG was written specifically to for SDL. This should mean that SMPEG should fit within the SDL API, and so combining the two in a project is made simple. SMPEG is chosen also because Pygame uses it to support MPEG playback.

A basic mpeg player would simply setup the display context and sound mixer in SDL then provide the apropriate hooks in SMPEG to use them. This is really just a matter of passing the screen pointer given by SDL then passing to SMPEG, similarily for audio. Once this is set, Pliant would simply invoke the process then totally surrender execution to the C compiled routines. Neither Pygame nor Pliant+SDL would perform any processing while a movie is playing.

This objective will not prove Pliant's worth over Pygame in this respect, but rather that Pliant can "glue" together routines from two or more libraries, using its own language. Pygame is a module compiled in C, so its easy to link with multiple libraries written also in C. The Pygame module likely wraps much of the process of setting up the display and connections with SMPEG into one simple class with a few methods. In Pliant, its expected that another set of C data structures and prototypes that are in SMPEG will need to be imported into the Pliant language. Then only can a wrapper module be made to make playing movies in Pliant as easy and automatic as in Pygame.

3.8 OpenGL

OpenGL is a graphics library for creating real-time 2D and 3D graphics. OpenGL is also used a standard API for graphics accelerators. This means that programs that use the OpenGL will be able to display and render 2D and 3D graphics fast, using the features availible by the hardware. Many games today use either Microsoft's Direct3D or OpenGL.

OpenGL[12] is actually owned by Silicon Graphics Inc, and despite its name the library isn't open-source software as specified by the Open Source Intiative[13](OSI). Another library, Mesa3D is a true open-source implementation of the OpenGL API,

[12]More information about OpenGL at `http://www.opengl.org`

[13]More information about OSI at `http://www.opensource.org`.

and is available on all platforms, especially Linux. So while Mesa3D is the actual graphics library, the term OpenGL refers to the API it implements.

SDL has built-in support for OpenGL. For example, SDL has a bit-flag that can be set upon initializing the screen. This makes and registers a device context with OpenGL. Then like in the GLUT (OpenGL Utility Toolkit), all gl* functions will begin to use and draw onto the SDL screen. This makes using OpenGL with SDL painless because SDL takes care of all the intialization, windowing and clean-up.

Pygame currently doesn't directly support OpenGL. Python does have bindings to OpenGL through another module called PyOpenGL. While PyOpenGL works and is actively being developed, only few applications have been made of it. And there has been even fewer that used Pygame and PyOpenGL together. Also, mailing-lists suggest that this combination doesn't quite work well.

Pliant however may have fewer problems combining with SDL and OpenGL. Because of Pliant's close intimacy with C, it may be possible to link SDL and OpenGL the same way C does. Pliant can use SDL and OpenGL because they both are DLLs. The objective is then to be able to easily make OpenGL available in Pliant+SDL.

If this is succesful, then it will proceed with performance comparisons with C. Its expected that while C will still be faster, but not significantly more than in Pliant. The test will consist of a rotating 3d object, and some geometric transformations implemented both in C and Pliant.

4 Requirements

The requirements for this thesis will try to make use of free software and already aquired hardware. This thesis should cost only in terms of time and not money[14].

4.1 Software

Linux
: Linux, any distribution with X11R6 will be used as the main development environment.

Windows
: Windows 9x/NT/2K/XP to test portability in Windows. All software should be able compile on both platforms.

GCC 2.95.4
: For compiling SDL, Pliant and Python. Windows version of GCC available in Cygwin or Mingw.

Python 2.1
: Python 2.1 distribution for both Linux and Windows.

Pliant 75
: Pliant 75 or later. For both Linux and Windows.

[14]The busineses arguement that time is money does not apply here, this project is not for profit.

SDL 1.2
SDL 1.2 libraries and include files, compiled from source distribution on both Linux and Windows.

OpenGL OpenGL libraries are often provided by graphics accelerator manufactures. Windows systems come with software implementations of OpenGL. On linux, Mesa3d can be used if there are no other OpenGL libraries available.

4.2 Hardware

There is currently one home machine, which meets the requirements and will suffice. Ideally, there would be one system to do research on, and another to develop. Software development isn't typically dangerous to a system but its usually a good idea. Bugs in software can cause machines to crash or hang. A second system is also useful as a back-up system. The basic hardware requirement for for Pliant+SDL is a soundcard and a 3D graphics accelerator. Below is an example of a suitable thesis machine along with cost[15], should one be considred for this project.

This machine is based around AMD's architecture because its a more affordable substitue for Intel. This machine is assembled from seperate parts and is an affordable configuration that meets the requirments well. Also, its designed to be silent as possible. Current computers often make excessive noise and can become distracting while working.

Part	**Description**	**Cost**
CPU	AMD Athlon XP 1800 Mhz	$139.00
Heat Sink	Zalman "Flower" Silent Cooler (20dBA!)	$74.00
RAM	256 133Mhz SDRAM	$49.00
Mainboard	ASUS A7S333 Socket A SiS745 (ATX)	$99.00
Video	ASUS v8170DDR GeForce4 440MX 64MB (no fan!)	$139.00
Sound	SoundBlaster Live Value 5.1 (OEM)	$49.99
HDD	40 GB Western Digital Ulta-100 7200RPM	$119.00
FDD	Panasonic 3.5 floppy drive	$19.99
CD-ROM	Sony 52X CD-ROM Drive (OEM)	$44.99
Case	Enlight Mid-tower w/ 7 bays.	$59.99
Power	Quietpc.ca ATX Ultra-Quiet PSU 300W (26.4dBA)	$128.00
Keyboard	104 Keyboard PS/2	$19.99
Mouse	Logitech Optical Wheel Mouse (OEM)	$29.99
Speakers	Generic 120W Stereo Speakers	$9.99
	Total:	**$982.92**
Monitor	Viewsonic P75F 17inch Flat Screen Monitor.	$339.99
	With Monitor:	**$1322.91**

[15] Prices based on current prices as found on the internet.

5 Expected Results

Ultimately, C/C++ is expected to be the performance leader. C is as low-level as high-level programming languages come. Years of implementation prove that C translates well into optimized assembler. C/C++ will likely continue to be the defacto standard for programming, high-performance or otherwise.

Python with Pygame is expected to give the least amount of performance. Python uses an interpreter, and its theoretically impossible for it to match compiled code. Python's power lies not in its speed, but its flexiblity and ease of use. Infact, its expected that Pygame will be better designed and easier to use than Pliant+SDL. Pygame has undergone more than a year of development and has a large following of contributors. Also, Python's object-oriented features make program organization conceptually easier.

Its expected, or rather its hoped that Pliant+SDL will prove to be a worthy comprimise of performance and ease of use. Pliant+SDL programs will have a long initial phase before actually executing. Once compiled, Pliant+SDL programs are expected to perform much faster than Pygame. Pygame may in some or most cases approach performance levels of C. Also with precompiling, Pliant+SDL programs may load significantly faster.

6 Conclusion

This project's purpose is to study and examine the possibities of using Pliant in videogames and multimedia applications. Its purpose is to develop any required bindings, prototypes and documentation to provide or demonstrate how Pliant can access functions in the SDL library. The result is Pliant+SDL and it will model its API to match the ease of use that Pygame offers.

Some objectives include making special effect demos, a small game and movie player. Comparisons with C and Pygame will be made along side the study. Pliant+SDL will also be tested on both Linux and Windows platforms, to study its ability to cross-platform.

Its expected that Pliant+SDL will offer programmers the ease of Pygame, with the performance of C. And its hoped that this thesis and Pliant+SDL can offer ideas for making better tools for prototyping high-performance video games and multimedia applications.