TinyGL: A Graphical Framework For Teaching Object Oriented Design and Programming

Joseph Fall (jfall@capcollege.bc.ca) and Jason Madar (jmadar@capcollege.bc.ca)
Computing Science Department
Capilano College

Abstract

The first year computing science curriculum at Capilano College is split into two courses: the first course (Comp120) focuses solely on top-down design and procedural abstraction, using C++, but without classes. The second course (Comp125) serves to develop and sharpen the students' abstraction and design skills, and provide a thorough exposure to object-oriented design and programming, also in C++. Comp125 is based on the CS1.5 model described by Michael Kuttner at the WCCCE conference in 2003. We have been offering this pair of courses since 1999, with successful outcomes similar to those described by Mr. Kuttner.

In order to make the course material more relevant and interesting for the students, we adopted a new approach to teaching Comp125 in the summer of 2002. We make use of a very simple graphics display package (TinyPTC), with a 3 function API (open, refresh, and close). Emphasizing the role of abstraction, layered architecture, and OO programming, the course builds on this simple API a basic graphics library, a set of animation classes, and, ultimately, a simple, but complete 2D video game (e.g., Pong, Space Invaders, Asteroids, Tetris, etc.) Students build their video games in a series of 4 assignments. First, they build an OO wrapper for the procedural TinyPTC API. This step abstracts the screen buffer and produces a Screen class, which the students find easy to think of in OO terms. Second, they build a basic vector graphics library (TinyGL), so that simple geometric shapes can be manipulated and drawn on the Screen. Third, they develop an OO design for a simple video game, based in TinyGL. This assignment serves as an introduction to OO design principles and introduces some of the basic design tools, like class and collaboration diagrams. Finally, students implement their game designs to produce a working system. This represents a significant and rewarding achievement for the students, as it is their first moderately large software project. In support of the assignments, the students also complete a series of 10 lab exercises. These exercises are used both to serve as practice for the concepts covered in the lectures, and also to introduce the concepts required for developing an interactive game.

Students' response to this approach has been very positive and encouraging. The students are more engaged with the course material, they appear to better understand the importance of good abstractions and interfaces, they come to an understanding of objects and data modeling quickly and with good intuition. Furthermore, without formally presenting the more advanced topics, the students seem to gain an appreciation of important software engineering principles such as code reuse, software architectures, and design patterns. Best of all, they have fun and really enjoy the course. Many students exceed the requirements for their final projects, adding colour, images, and extra functionality to their games. The approach has allowed students to be very creative, to explore a number of interesting computer science breadth topics, and yet provides enough structure that the ultimate objectives of the course are kept in clear focus.

Introduction

This paper describes our experiences using vector simulation and graphics programming as a framework for teaching OO concepts to first year computing science students. These ideas were initially developed in the summer of 2002, and have been successfully applied and refined over 3 subsequent offerings of the course. The basic idea is to have the students construct a basic graphics library, and then use it to design and build a simple 2D video game, using a layered-architecture approach. After presenting the course objectives and a bit of background, we'll provide a brief overview of the system we've been developing. Then we will discuss how the primary learning objectives are addressed by this approach, along with some additional opportunities for integration with computing and other science topics that naturally arise from the materials. The remainder of the paper will focus on the key insights and successes, along with the problems and potential pit-falls of this approach.

CS1.5 : Comp125 -- Fundamentals of Computing Science

Comp125 is the second in a pair of first-year coursed offered to computing science students at Capilano College. The course is based on the CS1.5 model described by Michael Kuttner at the WCCCE conference in 2003 [1], and we direct readers to his excellent paper for the rationale behind this model. For our purposes, it is important to understand only that students entering comp125 have already taken an introductory course in problem solving and programming (in C++), with a strong emphasis on algorithm development, procedural abstraction, interface specification, and top-down design.

The broad objectives for comp125 are twofold: (i) "to introduce the fundamental concepts of data abstraction and object-oriented programming, and to apply these concepts in the design and development of re-useable software modules"; and (ii) "to broaden the student's foundation in their knowledge of the discipline of computing science". The former goal emphasises the role of abstraction, interfaces, re-usable abstract data types (ADT's), component testing, and layered architectures in software development. The later is intended to introduce students to a variety of CS topics, including data structures, software engineering, numerical programming and simulation, computer graphics, analysis of algorithms, memory management, etc. The course also fulfils a tertiary purpose, to further develop and sharpen the students' problem solving, and programming, and design skills, which are typically still very weak after CS1. The course lectures are supplemented by a 2-hour lab exercise each week, and students must also complete a set of programming assignments to apply the techniques learned in class and the labs.

We have been using the "CS1.5 model" since 1999. Up until 2002, the course was structured around a set of more-or-less independent modules, each designed to deliver on a set of specific objectives. So, for example, the students worked on a rational number class to demonstrate re-useable ADT's and operator overloading; they implemented a simulation of an airport to demonstrate the use of queues in simulation modelling; and they designed and implemented the classic Solitaire card game to demonstrate OO design and the linked list implementation of a stack. While we found that the course was very successful, this "piecemeal" approach we were employing seemed to have a couple of limitations:

- Since each problem is developed anew, from scratch, there is simply not enough time to fully explore the subtler concepts and broader implications raised by each example, particularly the kinds of interface and design problems that only arise during integration within a larger system.

- The students do not get an opportunity to work on a reasonably large software project. This limited the meaningful application of re-use and layered architecture, thus weakening the students understanding of these fundamental concepts.

In 2002 we started developing a new approach to teaching comp125 that addresses these limitations by structuring the entire course around a single project. This project revolves around two primary activities: (i) the construction of a suite of vector graphics classes and simulations (which serves as the primary mechanism for meeting the course objectives); and (ii) the design and implementation of a 2D graphical video game (which serves as a strong motivator to capture student interest). Students are provided with a simple API for a graphics window that is refreshed from a software buffer. They construct a set of vector graphic and animation class libraries on top of this simple platform, leading naturally to a hierarchical, layered architecture, upon which they develop their game.

The Vector Graphics Simulation Framework

In this section we will briefly review how the project is developed through the course materials. For the purposes of clarifying this discussion, we present the development of the project in four distinct phases. In fact, the sequencing of the materials is more complex (see appendix I for details), and all four phases are largely developed in parallel. The lectures, labs, and assignments often focus on the development of different modules, or different aspects or applications of a module. Thus, in most cases we will not distinguish between materials presented in lectures, those completed as lab exercises, and those given as assignments.

Furthermore, many modules are introduced and developed in an independent context, and then integrated into the project. So, for example, a Projectile class is developed in the context of modelling the trajectory of a javelin, and is then re-used to form the basis for a Ball class (e.g., for Pong). These alternate contexts are not documented here in favour of a clear exposition of the main project. Nonetheless, this process is very important because it allows students to develop and test each module independently, and also reinforces the primary lessons about abstraction and re-use.

Phase 1: an OO Wrapper for TinyPTC

TinyPTC [2] is a simple, open-source graphics display package, written in C, that supplies a 3 function API to allow one to open and close a graphics window, and update the window contents from a software screen buffer. This screen buffer is represented implemented as a one dimensional array of 32-bit unsigned integers, with each integer representing the color of each screen pixel.

The primary motivation in this part of the project is to develop a screen "wrapper" class to bundle together the screen buffer, TinyPTC API, and basic rendering algorithms.

A graphics Buffer class is developed to provide basic image and line rendering abilities. Students develop their own line rendering algorithms based on the point-slope formula, and are then exposed to Bresenham's classic. This class is first implemented using a static array. Later in the term, students develop a dynamic array class (CVector) and then inherit from it a 2D array (CMatrix), that provides the mapping from 2D coordinates to 1D indices. The graphics Buffer class is then re-implemented based on CMatrix, to provide for a dynamic screen size, and to illustrate how a well-designed class can be completely re-implemented without changing its interface.

Finally, a screen class (CScreen) is developed to encapsulate the graphics buffer and TinyPTC API. This class provides clients with the ability to set the background, draw shapes and images, and refresh the display. The screen class that emerges from this work has a clean, consistent, easy to use interface. Because this class is so much easier to work with than the raw TinyPTC API, students come to an early appreciation for the power of good abstraction, encapsulation, classes, and OO programming.

Phase 2: TinyGL : A Homebrew Graphics Library for TinyPTC

In this phase, students develop and implement a set of classes to represent points, lines, rectangles, triangles, and circles in 2D space. Each object implements a basic set of operations, such as Centre, Move, and Rotate, in addition to a Render member, to render itself in a graphics Buffer as a series of line segments. The screen class is augmented to include an overloaded Draw function for each shape.

Once complete, students are aware of the duplication of effort required to implement these similar classes (i.e. the draw method for every shape class is actually very similar), and this is used to motivate the need for good OO design. At this critical junction, without formally presenting the concept of design pattern [3], the “Composite” design pattern is used as a solution to the problem. A poly-line class (CShape) is developed to serve as an abstract base class, and the geometric objects are re-implemented to inherit from CShape. Since this re-implementation is largely a matter of removing methods from the derived classes, students are made keenly aware of the importance and role of inheritance in facilitating re-use. Furthermore, all of the overloaded Draw methods in CScreen can be replaced by a single Draw(CShape&) method, illustrating polymorphism. Because the Circle class must override the basic poly-line Render algorithm provided by CShape, this also serves to introduce virtual functions and dynamic binding. Figure 1 shows the design of the CShape class hierarchy.

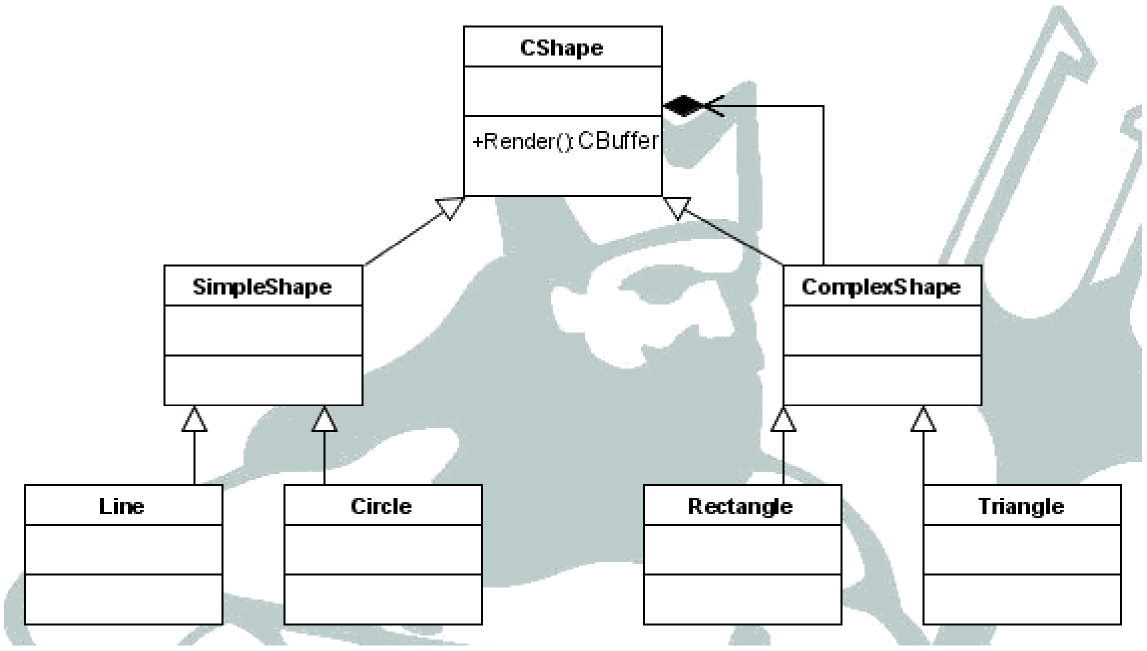

Figure 1 - Using the Composite Design Pattern in TinyGL

Finally, students are also introduced to raster graphics. They are provided with an open-source C++ class, CxImage [4], which provides a simple interface to load various image formats from file. (CxImage also provides extensive capabilities for image processing and transformation, but these are not generally used.) This part of the project provides an ideal opportunity to discuss efficiency issues, since naïve algorithms and large images noticeably affect the system performance. A retrofit to the CScreen class allows for efficient display of background images, and a new constructor to create a screen tailor sized to a background image.

Phase 3: Simulation, Animation, and Data Structures

This part of the project begins with some useful classes to encapsulate the representation of time presented by the <time.h> standard library. A Clock class provides the SystemTime in decimal seconds, as well as capabilities to Pause and Unpause the passage of time. This class illustrates two important concepts: (i) device interfaces (objects that change state due to external processes); and (ii) static members. Since time represents a truly global resource, students understand why it is acceptable to treat Clock members as global. Thus, simulation objects based on the Clock can move in "real time", and can be synchronised and paused as a unit. Two useful abstractions are derived from the Clock, an egg-timer (CTimer), and a stopwatch (CStopWatch) , both of which provide useful metaphors for tracking elapsed time. These classes also provide a reasonable example of private inheritance.

The projectile class (CProjectile) forms the base class for objects in free motion. Two implementations for this class are examined: the first simulates the differential equations describing change in motion (easy for the students to understand), whereas the second uses the solution to these equations to compute the current Position of the projectile. This provides for an introduction to discuss numerical integration and simulation using a relatively intuitive, and meaningful example where an analytic solutions is readily available. This class also provides an excellent example of composition and layered implementation, because its final implementation uses a CPoint (to represent its Launch position), a CVelocityVector (to represent its Launch trajectory), and a CStopwatch (to track elapsed seconds since Launch). Once the basic projectile is working, gravity and drag are added for completeness.

Although a point class has been previously developed, a separate vector class (CVelocityVector) is introduced to provide efficiency (stores vector in polar rather than Cartesian co-ordinates) and better abstraction (interface uses velocity and vector terminology). This vector class encapsulates many of the algorithms required for collision detection, including IsOrientedTowards(aPoint), DistanceToIntersect(aLine), and Bounce(aLine), which returns the vector that results from reflection off a flat surface. This provides many opportunities to explore interesting vector algorithms.

The CBall class is simply a projectile with a circular shape. Students tend to try to inherit the ball class from CCircle instead, which results in a poor design. This leads to a good example of how important it is to get the inheritance hierarchy right. Students develop a series of increasingly complex simulations using the ball class, and use it to develop and test some basic collision detection algorithms. Finally, they produce a container class (CContainer), which is a rectangle that can contain balls within it (bounces them off its interior). This class serves as the walls for a court (e.g., in Pong), and also as the base class for CExcluder, a rectangle that excludes balls (bounces them off its exterior). This serves as another excellent example of the power of inheritance and dynamic binding, since the excluder simply overrides the containers collision detect algorithm to bounce incoming balls rather than outgoing ones. The excluder class serves as a base class for paddles, blocks and other game pieces that require a collision detecting bounding rectangle.

Finally, the students also develop a set of data structures. In addition to the dynamic array classes (CVec and CMatrix) mentioned above, a generic collection class (CCollection), representing an unordered collection of homogeneous elements, is studied in detail. The collection class reinforces the distinction between internal and external representations for data structures, as it is first implemented as a dynamic array, and then the same interface is re-implemented with a linked list. These data structure classes are also used to introduce the idea of type-independent structures, and, eventually, template classes.

Phase 4: Design & Implementation of a 2D Video Game

Based on the class libraries developed in the previous phases, we have completed designs for games similar to Pong, BreakOut, Space Invaders, Asteroids, and Tetris. Typically, one game is selected to serve as an in-class example, while another is selected by the students to design and implement on their own. In general, only a description of the game is provided, and the students must complete the requirements analysis as part of the design exercise. This provides an opportunity for students to use their creativity to design the game details; it allows for students to self-determine the degree of difficulty they wish to tackle; and it virtually eliminates cheating, since each game turns out quite unique.

Students work in teams of 2 to produce an OO design for the game. This exercise provides students with an opportunity to apply all they have learned about interfaces, inheritance, and composition. We do not expect the students to produce great designs here. In fact, the primary, if unstated, goal is to demonstrate the difficulties of design, and the degree of effort required to fully think through a complete, functional design, even for a relatively staight-forward problem. For this reason, students are required to revise their designs as they implement their games (and thus discover the variety of design errors!), and re-submit their initial and revised designs with the final project.

Students are encouraged to use some simple design tools, including UML use-case, class, and collaboration diagrams. Similar to their introduction to design patterns, the diagramming techniques are not formally covered, they are just introduced casually, as techniques for visualising the design relationships.

Figure 2 shows a typical design for a "generic" space game with multiple "enemies" (in this case Balloons) and multiple "missiles" (in this case Darts). Although not strictly required, students tend to re-use the Game-View-Player association, shown here, in almost all of their designs, without encouragement. However, they do need some extra encouragement to create classes to manage collections, as they otherwise tend to confuse the data structure storing the collection with the abstract operations on that collection (e.g., Display, Collide, etc.). Some students fail to recognise the need for collections at all. The other game piece classes (such as Ship, Dart, Paddle, Block, etc.) are fairly obvious and students tend to do an adequate job of identifying and designing reasonable interfaces for them.

Key:	◆	"**Has-a**" relationship: object with diamond is "made of" object(s) pointed at
	▽	"**Is-a**" relationship: object inherits from base-class object pointed to with arrow
	----	**Strong Association**: objects are work closely together to solve problem

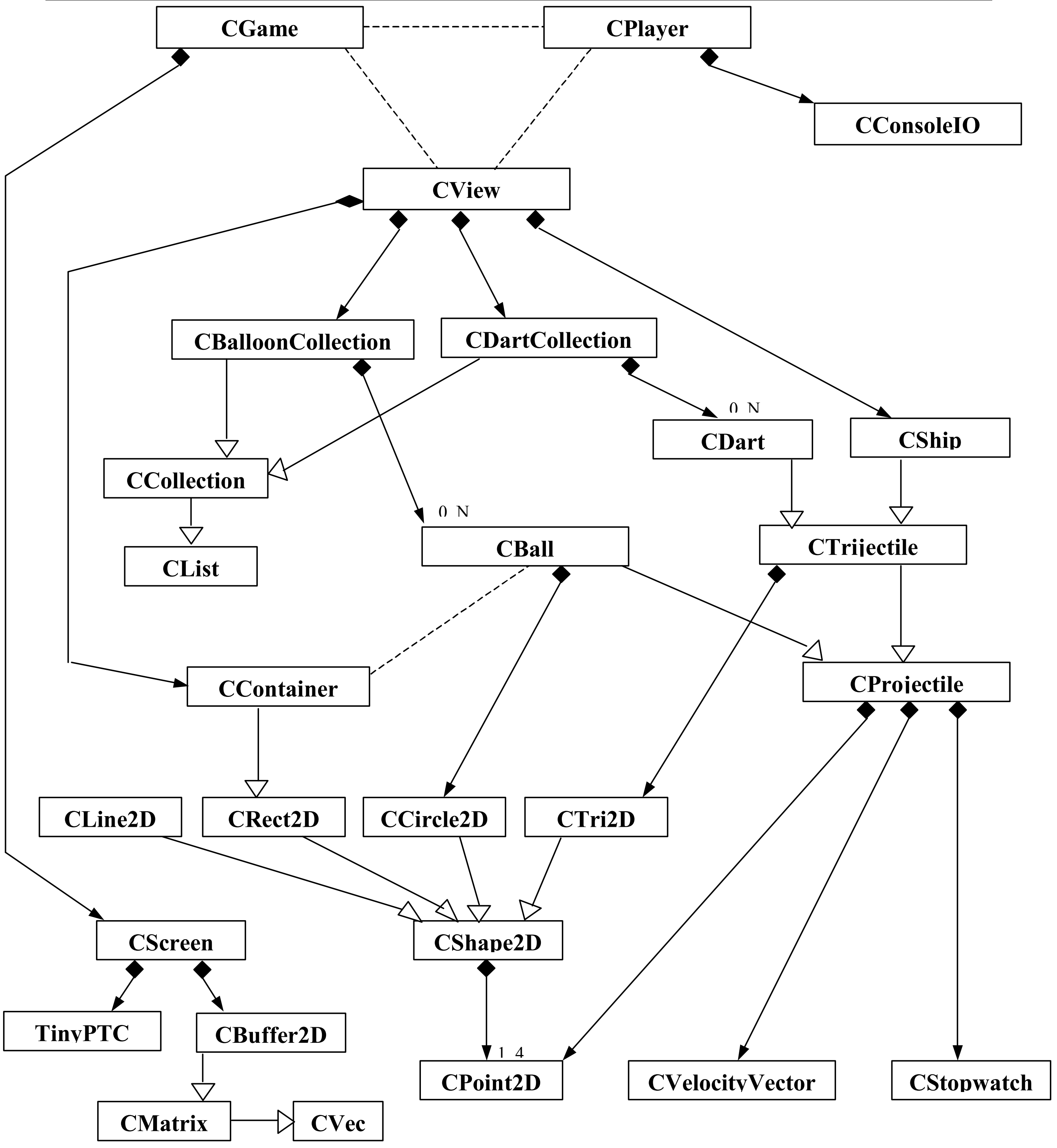

Figure 2 - Class Diagram (for generic "space-type" game, e.g., Asteroids or Space Invaders)

Learning Objectives

As stated in the introduction, the course has two primary objectives: (i) "to introduce the fundamental concepts of data abstraction and object-oriented programming, and to apply these concepts in the design and development of re-useable software modules"; and (ii) "to broaden the student's foundation in their knowledge of the discipline of computing science". In this section we will summarise how the project described above meets these two objectives, along with an some of opportunities for further integration with other disciplines.

OO Programming and Design Objectives

There are no difficulties in meeting these basic course objectives with this project. Here we simply list the key objectives for reference. (Details on the specific objectives met by each of the software modules are provided in Appendix I.) By the end of the course, we expect students to be able to:

- design, implement, and test programs of an intermediate size and level of complexity;
- use data abstraction to design and concisely document the interface for an ADT;
- choose an appropriate representation and implement an ADT from an interface spec.;
- understand the difference between the internal and external representations of an ADT;
- design a test driver and an appropriate suite of test cases to independently test an ADT;
- use and implement some basic data structures such as lists, stacks, and queues;
- explain and use the principles of encapsulation, information hiding, layered architecture, component testing, and re-use;
- develop abstract, re-useable, hierarchical class libraries in C++;
- correctly use pointers and manage dynamic data in C++;

Fundamentals of Computing Science? Meeting the Breadth Objectives.

A major strength of this project is the variety of opportunities it presents to illustrate a range of computing topics. We have found that the project actually facilitate many of the breadth objectives better than the piecemeal approach used previously. The following is a short list of some Computing Science topics that are closely related to particular course materials.

- ***Data Structures***: Students can see the utility of studying data structures because they so clearly facilitate the implementation of the game. Perhaps more importantly, they get several good examples of the distinction between the external logical structure and the internal physical structure. For example, in developing the CMatrix class, the students learn how to map a 2D logical structure onto a 1D physical structure; In developing the CCollection class, they see that the internal representation could be an array or a list.
- ***Analysis of Algorithms***: By allowing students to use naïve algorithms to re-draw large background images, it is easy to illustrate the importance of efficiency. Because images are naturally 2 dimensional, it leads intuitively to a discussion of O(n) vs. O(n^2) algorithms. Students are motivated to learn about efficiency because their games otherwise perform poorly.
- ***Software Engineering***: Due to the size and complexity of the project, there is an emphasis on interfaces, design, and layered architecture throughout. Students are introduced to

requirements analysis, UML, and design patterns, but simply as useful tools rather than with the formal rigour of an SE course.

- ***Graphics***: Students come away with a good understanding of colour models, graphics buffers, and how display devices work, and they can distinguish clearly between vector and raster representations. This provides ample background to discuss topics such as graphics hardware, graphics libraries (e.g., OpenGL), and 3D rendering algorithms.
- ***I/O*** : The user interface for the game requires a mix of buffered and unbuffered input, so students learn the difference between them. They develop their own I/O class based on the conio interface, thus gaining an appreciation for how the iostream classes work. We include a discussion of how I/O interrupts work to get keyboard and mouse input to the application.
- ***Data Representation*** : The RGB colour model provides an opportunity to examine a useful application of the Hexadecimal number system. To create different colours, students also learn about byte addressing, and bit-wise operations, including shift.
- ***Numerical Methods***: Several dynamic simulations are covered, each with an analytic solution. Thus, students are shown how simulation is really a numerical integration that approximates the solution to a differential equation, using examples they understand and can explore and compare. Students can then grasp the idea of equations with no analytic solution that require simulation, which leads to a discussion of the application of simulation modelling in various application areas.
- ***Animation & real-time programming*** : Since the game is actually a "real-time" vector animation, it provides ample opportunity discuss these topics.
- ***Event-driven programming*** : Typically, programs developed at this level wait for user input, but the animation loop does not allow for this. Thus, the Player class is designed to trap user input events and then call the appropriate method to handle the event. The View class is thus designed very much like an event-handler, with methods to take action on each user input event, in addition to any collision events. So, although there is no actual event programming here, in many ways it is easy to explain in this context because the students actually program both the event trap and handler (as opposed to, more traditionally, programming only the handler, with the events trapped "by magic").
- ***Multi-threading*** : Although the entire project is completed as a single thread, students tend to naïvely develop multi-threaded solutions. For example, students want a separate thread to monitor user input, or a separate thread to display an animated image. Because students are already receptive, this provides an excellent opportunity to discuss the difference between single- and multi-threaded applications, and raise some of the difficulties involved with multi-threading.

Opportunities for Integration

The vector simulation framework provides many nice opportunities for integration with materials from other disciplines. We briefly discuss here a few examples that either arise naturally from the course materials, or we have incorporated for interest sake.

- ***Math***: Students are often surprised by the amount of math required for the project. Even the basic shape algorithms require some tricky geometry (e.g., find the centre and bounding circle for a Triangle), and the collision detection requires a good dose of vector algebra (e.g., orientation, line-line intersection, interior of a rotated rectangle). As discussed above, there are also opportunities to discuss differential equations and their solutions.

- ***Physics***: The projectile class gives students a practical application for Newton's laws of motion. Once implemented, the library can easily be used to simulate orbiting bodies, escape velocity, etc.
- ***Biology***: The easiest simulation to understand is likely an exponential model of population dynamics. We use this to introduce the idea of dynamic simulation and objects that "change by themselves" (in this case, the population object grows itself in simulated time). The framework can also be used to produce a fun expose of Darwinian evolution. A simple application has a container with a collection of balls. One ball is launched into the container (added to the collection) each 5 seconds. The size of each new ball is selected at random from a distribution with mean equal to the average size of all balls currently in the collection. When two balls collide, the smaller ball is removed from the collection. These two simple rules put a strong selective pressure on the balls to get larger. Not only does this serve as a nice programming example, an interesting simulation, and a clear demonstration of the principles of natural selection, but it also shows how programs can produce feedback and emergent behaviour -- in this case the desired effect, but not always!

Conclusions: Lessons and Pitfalls.

The approach to teaching CS1.5 described above has been immensely successful, both in terms of providing a sound mechanism for meeting our course objectives, and in terms of generating positive student feedback. Unfortunately, this approach was not designed, it just evolved, and thus we do not have any formal analysis of its impact on outcomes, and can offer only anecdotal observations. We believe the course is generating more enthusiastic and positive evaluations from students. We have received many comments like "this is the best course I have ever taken" and "all courses should be like this". Students appear to be very engaged with the course materials, and spend a lot of effort on their projects. We also believe that this approach is preparing students better for their data structures and software-engineering courses in second year. Having seen and experienced the entire design and development process for a substantial project, we think that students will have a greater appreciation for SE concepts, tools and techniques. We leave you with a short discussion of the key successes, potential pitfalls, and some lessons we've learned along the way.

Successes

One of the keys to success is having a project large enough that the students can not possibly hold all the implementation details in their heads (as they otherwise seem to try to do!). This forces students to think abstractly about the class interfaces, and to create layered implementations. On the flip side, we think it is equally important that the students are intimately involved in the design and implementation of all components in the system. This ensures that they understand how the whole system works, and thus that there are no "magic" components.

A few other brief, but significant successes include:

- students find the materials challenging, but are able to immediately identify the relevance of each new topic as it is applied in context to the project.
- vector graphics simulation seems to lend itself naturally to an OO solution;
- the shape inheritance hierarchy is intuitive, and provided identifiable benefits to students;

- takes the "magic" out of graphics applications, and illustrates what a GL is;
- collision detection provides an excellent example of a non-trivial problem;

Pitfalls

There are risks with this approach, and the potential for a complete disaster. Here are a few potential problem areas we have identified.

- Fear factor : The idea of graphic programming overwhelmed a few students at the start. Students need to be eased into the material, preferably with each new concept introduced in an independent context, and then applied to the graphics project once understood.
- Potential to get out of control: There is more here than meets the eye! The project must be managed carefully to ensure it does not overwhelm students' ability to cope. In addition, one must be aware that there are a few tricky algorithms that appear simple (e.g., center of a triangle), and thus could cause students to waste a lot of time;
- Difficulties using debugger with "real-time" moving objects: Since the projectile class moves "on its own" in real time, it can be difficult to trace in a debugger. Students need to be shown how to bracket break-point areas with Pause and UnPause so that simulation objects based on the projectile pause while the debugger is in step mode.
- Sequencing materials: a few topics couldn't come soon enough! We found pressing need for pointers and inheritance well before they were scheduled. The order of topics was re-arranged to keep up with project requirements, which meant skipping around in text book much more than in past terms;

Lessons

Finally, here are a few lessons we learned the hard way about things to do and things to avoid:

DO : delay announcing which game will be used for the final project
When students don't know what game they will be programming, they are forced to develop flexible, re-useable objects. They seem to appreciate the strength of re-use and layered architecture when they see how several different games can be developed on top of the same platform of class libraries. This term we allowed students to vote for which game to implement for the final project, which seemed to provide additional incentive.

DO : provide timely access to algorithms
The project uses many classic algorithms from a diversity of fields, including graphics (e.g., rendering), geometry (e.g., triangle centre), and vector algebra (e.g., intersection). Since there is plenty of opportunity for students to solve problems and develop algorithms, we develop some of the more complex algorithms with the students directly, and allow students to use web resources (such as http://mathworld.wolfram.com), provided they cite their sources.

DO : use UML diagram techniques
We have found that students find Use-case, Class-, and Collaboration-Diagrams intuitive. We use them freely to illustrate and gain an understanding of the design, without worrying too much about the formal semantics.

DO : discuss design criteria
The design criteria for this project are consistent with the course objectives, but not really with standard graphics programming. There is a potential for students to develop an inappropriate model for graphics programming. To avoid this, one should emphasise that the design criteria for real graphics programming differ from those used in course.

DO : provide class libraries at start of game implementation
After allowing students to struggle with developing their own class libraries, we provide a working, debugged version of the TinyGL, Animation, and Data Structures class libraries to serve as a solid platform for their projects.

AVOID : examining SE process models, etc.
This is not a course in software engineering, and so the instructor should choose the design and process methodologies. While the students are made aware of the design process, and take part in designing subsystems, the process models, design tools, and system design are kept transparent, so that the students can take part in the process, without having to understand exactly how it all works, or what the alternatives are.

AVOID : providing incentives for "extra" features
There is a great potential for students to get carried away with creating "fancy" graphics, user-interfaces, and other features for their games instead of focussing on the design and interface details. Warn students against this and do not reward such "extras". A few good students were unable to complete their final project due to wasting time on unrequired features.

Appendix I : Class libraries

Class Libraries

The tables below describe the 4 major class libraries developed in the course. We also provide some of the specific C++ programming and OO concept objectives delivered by each module.

TinyGL: a tiny graphics library.

Module	**Description & primary objectives**
TinyPTC API	Review modules, interfaces, and arrays. Familiarise students with the TinyPTC API, primarily in preparation for developing the Screen class, below.
Screen class	Develop a "wrapper" class that hides the TinyPTC API and its associated software screen buffer (see data structures below) behind a single, consistent interface.
2D Points	Introduction to ADT's and C++ classes. Includes: implementation of math operators; basic 2D vector algorithms; distinction between internal and external representations (Cartesian vs. polar co-ordinates)
2D Lines	First real use of composition to create a more complex object. Introduction to vector graphics, and line algorithms.
2D Poly-Shapes	Serves as base class for open or closed polyline shapes. Demonstrates distinction between abstract representation (sequence of line segments) vs. concrete representation (sequence of verticies). Introduces concept of abstract base class, virtual functions, and protected members. Includes algorithms for computing Bounding Rectangle and Bounding Circle for a shape.
Rectangles & Triangles	Basic graphic objects that inherit from the shape class. Develop algorithms to compute Centre of object and to determine if a point is Contained within it.
Circles	Inherits from Shape, but overrides the Render method. Illustrates function overriding, and develops rotation algorithms. Includes algorithms for determining circle-point, and circle-circle intersections used for collision detection.
Images	Introduction to raster graphics and standard image formats. Employs open-source C++ class CxImage (http://www.codeproject.com/bitmap/cximage.asp), which provides simple interface to load various image formats from file. Includes retrofit to Screen class to allow for efficient display of background images.

Animation: a suite of basic vector simulation and animation classes.

Module	**Description & primary objectives**
Random Numbers	Introduce students to pseudo-random number sequences and simple generating algorithms and transformations.
Clock, Stopwatch and Timer	Three very simple, yet useful, ADT's that hide real-time management. Serve as examples of device interface (clock), static class members (SystemTime & Pause), as well as abstract interfaces and private inheritance.
Velocity-Vector	Polar co-ordinate representation for 2D vector. Emphasises efficiency in representation, and conversion between data types. Includes algorithms for computing Inverse and Reflected (Bounce) vectors.
Projectile	Simulates an object in motion in 2D space. A great example of composition (composed of 3 simpler classes to produce complex behaviour). Demonstrates

	dynamic simulation, numerical vs. analytical solutions. Forms base class for many of the animation objects.
Ball class	A projectile with a circular shape. Emphasis is on getting inheritance relationship right (i.e., seems natural to inherit from Circle, but this leads to poor design).
Tri-jectile	A projectile with a triangular shape. Used as a base class for space ships, missiles, etc. Introduces concept of a bounding circle for easy collision detection.
Container	A rectangle that can contain a ball within itself. Introduces basic circle-rectangle and vector-line collision detection algorithms. Provides goo opportunity for discussion on collaborations between classes.
Excluder	An "anti-Container" for bouncing balls off exterior of rectangle. Useful for creating paddles and blocks or as a bounding rectangle for handling collision detection. Emphasis is on re-use and dynamic binding (overrides collision detection)

Data Structures: a set of useful collection classes.

Module	**Description & primary objectives**
Vector	Dynamic array class. Introduces dynamic memory management, deep vs. shallow copies, destructors. Forms base class for other data structures. Used to demonstrate "type independence" of data structure and class templates.
Matrix	2D interface to Vector class. Demonstrates mapping of 2D logical structure onto 1D physical structure.
Buffer	A software graphics buffer class for use in conjunction with TinyPTC. Initially coded with static array, later re-implemented as a dynamic Matrix to illustrate how implementation can change without changing class interface.
Collection	An unordered collection of elements. Introduces abstract interfaces for data structures with two implementations: dynamic Vector and Linked List
Linked List	Introduction to the classic singly linked list, used to implement a collection.

Game: a selection of game-specific classes

Module	**Description & primary objectives**
Player	Hides user interface from rest of system. Encourages students to build their system to be as independent as possible from any particular interface.
Game Pieces	Includes user-controlled objects, such as Paddle or Ship, and automated objects such as Asteroids, Missiles, or Invaders. Emphasises object modelling and the use of layered architecture using both inheritance and composition.
Game Piece Collections	Most games require a collection of game pieces. Students are encouraged to encapsulate the management of those collections in a separate class.
View and Game classes	High level classes that manage all of the game pieces (View or Court) and contain algorithms for playing a game (Game). Demonstrates the elegance and simplicity of abstract, high-level algorithms in a well-designed system.

Appendix II: Key Concepts & Fundamental Algorithms

One of the most exciting parts of this project (from an instructional perspective) is the opportunity to discuss, develop, and analyse some classic, difficult algorithms. In each case, students are encouraged to develop a solution to the problem, which appears deceptively simple. We then analyse a variety of solutions, including the classics. These examples clearly demonstrate the process of algorithm discovery and development, and provide a meaningful motivation for the analysis of algorithms and efficiency.

Graphics and Animation Concepts

Concept	**Description & primary objectives**
Graphic Devices	Familiarise students with the concepts of raster display devices, pixels, and colour models, focussing on RGB encoding (used by TinyPTC). Motivates need for bit-wise operations and an understanding of the representation of built-in types.
Rendering Algorithms	Familiarise students with algorithms for converting vector representations to raster. Emphasis is on efficiency and study of classic algorithms.
Animation Cycle	Familiarise students with the basic animation loop (Move, Detect Collisions, Draw, Refresh). Emphasise difference between single-threaded and multi-threaded algorithms (students seem to naturally think of multi-threaded solutions).
Collision Detection	Familiarise students with basic algorithms for detecting collisions between vector objects in 2D space. This notoriously difficult problem initially appears to students as a non-issue. Thus, it serves to illustrate the difficulties of spatial reasoning.

Appendix III: Chronological Sequence

There are dependencies in the sequencing of these modules, and so a rough chronology of the development sequence we currently use is also provided in the table below. Note that some of these modules are actually developed in several stages and this is not represented in the sequence below.

	Topic	**Module(s)**
1.	ADTs (I)	Point2D class, operators, and basic 2D vector algorithms
2.	Graphics Devices	Raster displays and the RGB colour model
3.	ADTs (II)	Clock, Timer, & Stopwatch classes
4.	Data Structures (I)	2D screen buffer
5.	APIs (I)	TinyPTC interface & Screen "wrapper" class
6.	Simulation (I)	VelocityVector and Projectile classes
7.	ADTs (III) Vector Graphics	Line, Rectangle, Triangle, & Circle classes Rendering and rotation algorithms
8.	APIs (II) Raster Graphics	CxImage class interface
9.	Dynamic Memory	Vector & Matrix dynamic array classes
10.	Console I/O	ConsoleIO class (buffered vs. unbuffered, echo vs. quiet)
11.	Inheritance (I)	Shape abstract base class -- inherit vector classes
12.	Inheritance (II)	Ball class
13.	Animation Algorithms	The animation loop using the ball class Collision detection algorithms
14.	Inheritance (III)	Container and Excluder classes
15.	Data Structures (II)	Collections (vector implementation)
16.	OO Design (I)	Preliminary game design (class diagram)
17.	Layered Architecture (I)	Game Pieces (e.g., paddle, ship, missile, asteroid, block)
18.	User Interface Design	Player class (use-case analysis)
19.	Layered Architecture (I)	Game piece collections (e.g., asteroids, blocks, missiles)
20.	OO Design (II)	Game and View classes (collaboration diagrams)
21.	Data Structures (III)	Collections (Linked List implementation)
22.	Implementation	Final project implementation

References

[1] Michael Kuttner, http://www.cs.ubc.ca/wccce/Program03/papers/Kuttner.htm, WCCCE 2003
[2] The TinyPTC library, http://sourceforge.net/projects/tinyptc
[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns. Addison-Wesley Pub Co. January 15, 1995. ISBN 0201633612.
[4] CxImage, http://www.codeproject.com/bitmap/cximage.asp
[5] Heddington, Mark R., and David D. Riley. Data Abstraction and Structures Using C++. Jones & Bartlett, 1997.
[6] Brookshear, J. Glenn. Computer Science, An Overview. 6th ed. Addison Wesley, 2000.