

Critters in the Classroom: A 3D Computer-Game-Like Tool for Teaching Programming to Computer Animation Students

Eike Falk Anderson* and Leigh McLoughlin†
The National Centre for Computer Animation
Bournemouth University



Figure 1: Sheep on “The Meadow”.

Abstract

The brewing crisis threatening computer science education is a well documented fact. To counter this and to increase enrolment and retention in computer science related degrees, it has been suggested to make programming “more fun” and to offer “multidisciplinary and cross-disciplinary programs” [Carter 2006]. The Computer Visualisation and Animation undergraduate degree at the National Centre for Computer Animation (Bournemouth University) is such a programme. Computer programming forms an integral part of the curriculum of this technical arts degree, and as educators we constantly face the challenge of having to encourage our students to engage with the subject.

We intend to address this with our C-Sheep system, a re-imagination of the “Karel the Robot” teaching tool [Pattis 1981], using modern 3D computer game graphics that today’s students are familiar with. This provides a game-like setting for writing computer programs, using a task-specific set of instructions which allow users to take control of virtual entities acting within a micro world, effectively providing a graphical representation of the algorithms used. Whereas two decades ago, students would be intrigued by a 2D top-down representation of the micro world, the lack of the visual gimmickry found in modern computer games for representing the virtual world now makes it extremely difficult to maintain the interest of students from today’s “Plug&Play generation”. It is therefore especially important to aim for a 3D game-like representation which is “attractive and highly motivating to today’s generation of media-conscious students” [Moskal et al. 2004].

Our system uses a modern, platform independent games engine, capable of presenting a visually rich virtual environment using a state

of the art rendering engine of a type usually found in entertainment systems. Our aim is to entice students to spend more time programming, by providing them with an enjoyable experience.

This paper provides a discussion of the 3D computer game technology employed in our system and presents examples of how this can be exploited to provide engaging exercises to create a rewarding learning experience for our students.

CR Categories: K.3.2 [Computers and Education]: Computer and Information Science Education—Computer Science Education

Keywords: Pedagogy, Programming, Visualisation, Games

1 Introduction

The crisis in science education and especially in computer science education, due to dramatically reduced enrolment numbers and high attrition rates [Carter 2006], has escalated to the point that it is now also reported on in the main-stream media [Koch and Mohr 2006; Ghosh 2006]. Computer science and related subjects suffer from a range of problems, many of which appear to be image related, possibly caused by a general misconception of computer science – sometimes confused with computer literacy [Beaubouef and Mason 2005] – and computer programming among prospective students.

In our experience of teaching on a long established computer animation degree with very strong technical components, we have found a great number of our students to be strongly biased against computer science related units, especially against computer programming. Our course, the Computer Visualisation and Animation undergraduate degree at the National Centre for Computer Animation (Bournemouth University) fits the description of the “multidisciplinary and cross-disciplinary programs” proposed to counter this trend and to increase enrolment in computer science related degrees [Carter 2006]. Our students find employment as technical directors in the digital special effects and computer animation industries and as technical artists in the computer games industry. To prepare them for these roles, our arts degree has a strong technical, computer science related component, and computer programming forms an integral part of the curriculum.

*e-mail: eanderson@bournemouth.ac.uk

†e-mail: lmcloughlin@bournemouth.ac.uk

Computer programming is an essential skill for software developers; however, even the fact that students are pursuing a computer science related degree is alone no indicator for the existence of motivation and it can be very difficult to interest them in the act of computer programming, the writing of software, itself. This is something we have observed among students of our course who come from very diverse backgrounds, often with little prior experience with computers. Motivation is a major factor in the success of students of programming, who can only improve their skills by practicing writing programs [Jenkins 2001]. We especially face motivational problems among those students who joined the course solely out of interest for its artistic elements and who consider the computing aspects of the course as a necessary evil. This lack of motivation and interest may be grounded in a general misconception of computer science and programming among prospective students who perceive it as boring (non-creative) and difficult (i.e. actually requiring work).

A lack of knowledge of the subject area may well be the underlying reason for this bias against computer science and programming. Among prospective students we have encountered candidates who appear intimidated by the technological aspects and the maths involved with the course we offer. Other candidates are overconfident, having taken an information technology course at school, which in reality is little more than a computer literacy course. As soon as they realise their mistake, the latter quickly lose interest once they have embarked on the computer science related course (or module), an observation also made by Beaubouef and Mason [2005].

A major source of frustration appears to be a lack of patience that we have observed among students from the “Plug&Play generation”. Real-world programming languages often require a lot of understanding, as well as a lot of work, before useful results are obtained. The students however expect to see immediate (and spectacular) results, often before they have learned enough to achieve anything remotely spectacular. Their programs often have unintended results, causing confusion, and once faced with difficulties (a recent study of which was made by Lahtinen et al. [2005]), their motivation suffers and they quickly lose interest in the subject, causing them to fall behind in their studies. In our experience this is a vicious circle, causing frustration and a further loss of motivation, leading to a downwards spiral and eventually ending in exam failure. We believe that an important step towards solving this motivational problem is to make the act of programming appear less alien to our students. The solution in this case must be a familiar environment for programming which is not purely a learning instrument, i.e. not too far removed from real-world systems to appear irrelevant, but which is both simple and close to real-world systems.

Having had a positive first hand experience of learning programming through use of a “Karel the Robot” [Pattis 1981] like program, it appeared only logical to try to use a similar approach for teaching our students. C being the language traditionally used in our undergraduate course, our solution is the C-Sheep programming language, a fully compatible subset of the ANSI C programming language [Kernighan and Ritchie 1988]. The virtual entities whose behaviour is controlled by C-Sheep programs are sheep, inhabiting “The Meadow” virtual environment. The 3D game-like virtual world of “The Meadow” is built on a proprietary game engine [McLoughlin and Anderson 2006] which incorporates a virtual machine for executing C-Sheep programs. By offering modern 3D computer graphics and effects more commonly found in computer games, our goal is to help students to realise that computer programming can be an enjoyable and rewarding experience.

2 Related Tools

Over the years, following in the footsteps of Logo and its turtle [Papert 1980], there have been many tools for teaching programming in particular and computer science education in general, a detailed overview of which is presented by Kelleher and Pausch [2005]. The game-like setting of “Karel the Robot” [Pattis 1981], its successors and related systems, as well as the success that they have achieved, make them especially attractive to educators who have to motivate students to spend more time programming. These systems can be counted in the family of mini-languages [Brusilovsky et al. 1997; Anderson and McLoughlin 2006] that provide a task-specific set of instructions and (sensor) queries which allow users to take control of virtual entities, acting within a micro world. This micro world provides a graphical representation of the algorithms used in the programs controlling the virtual entities. Their position and orientation within the virtual world provide visual feedback of the current state of the program, which is especially useful as many problems faced by novice programmers can possibly be traced back to an inadequate understanding of program state [Dann et al. 2000]. The aim of these introductory programming environments is to motivate students to take up programming, and to provide them with an enjoyable experience at the same time, by solving tasks in a sandbox environment in which the human player interaction is limited to the programming of the virtual entities that act out the solutions.

In fact, there are several examples of games that provide interaction through this method, such as the Java based Robocode [Li 2002], which has also been used in computer science education [Bierre and Phelps 2004; Hartness 2004], or the full 3D action game GUN-TACTYX [Boselli 2004]. Among these games the ones with 3D computer graphics are of special interest, as they are most likely to help with meeting the high expectations of the “Plug&Play generation”: The traditional mini-language systems are now severely outdated and the 2D top-down representation of their micro worlds (often restricted to ASCII characters in text-mode) is no longer enough – an assumption that is reinforced by the negative student reaction to the 2D Robocode system reported by Bierre et al. [2006].

We have chosen an interactive environment using 3D computer graphics to meet the expectations of our students. Only a few examples, such as Alice or the MUPPETS system [Cooper et al. 2000; Phelps et al. 2003], use true 3D graphics for their micro world representation. This is surprising since the 3D game-like representation of the program state created by the virtual entity is “intrinsic in the natural way to view the data itself” [Dann et al. 2000]. Both systems act on the same premise, but use a different approach to ours.

While it is based on the Java language (an earlier version used Python instead), the design of Alice explicitly tries to remove language complexity by hiding the language syntax from the user, allowing the “building” of programs using context-sensitive menus [Kelleher 2006]. While this can help with the teaching of pure concepts, this avoidance of syntax unfortunately does little for the learning of programming using a programming language.

Another approach was taken with the MUPPETS system [Phelps and Parks 2004], which provides a networked virtual world. This is rendered in 3D using modern games technology, shared amongst several students who create objects that can interact with each other. The students develop the objects using the Java language in a comprehensive IDE (integrated development environment) which is included in the virtual world interface.

The commonality is that by offering modern 3D computer graphics and effects more commonly found in computer games, their systems and our system aim to motivate students to take up program-

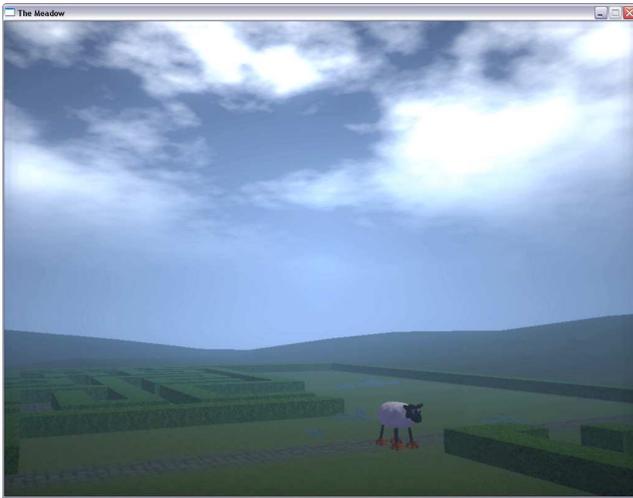


Figure 2: “The Meadow” virtual environment.

ming and to help them realise that computer programming can be an enjoyable task. While many of the available systems have been successfully used in the teaching of programming, unfortunately those that are more graphically appealing do not fit our requirements. Their choice of language and programming paradigm have made them unsuitable for our course, prompting us to develop our own solution.

3 “The Meadow” in a Game Engine

“The Meadow” virtual environment is the virtual world in which entities controlled by C-Sheep programs exist (see figure 2). This is enabled through a compact portable game engine (currently supporting Linux and Windows operating systems) which was designed specifically for “The Meadow”, yet it is flexible in design and offers a number of features common to more complex engines. In its current form, the engine achieves scene management through a scene graph data structure, which also acts to bind the system components together as a whole. The main game engine is itself subdivided into a number of modules and supporting structures, some of which we will cover in more detail. Management of the rendering context, sound output and user input are achieved through the Simple DirectMedia Layer [Latinga 2004]. The engine has a flexible GUI (graphical user interface) system, with basic window management, widgets and input/output methods. The system also contains a physics module based on the Open Dynamics Engine, which provides physically based dynamics and allows for collision detection and resolution [Smith 2003]. In comparison with some recent computer game releases, this feature list may seem unremarkable; however, it is quite extensive for a teaching tool.

3.1 Rendering Module

Rendering is achieved with a rendering engine using the OpenGL API (application programming interface) [Shreiner et al. 2005], supporting the programmable graphics hardware pipeline through NVIDIA’s Cg Shaders [Mark et al. 2003]. With the exception of the GUI system, the engine uses Cg for almost all of its graphical output, and most of the advanced features also rely on OpenGL Frame Buffer Objects, which provide a convenient method of rendering to texture. When the system is run on a machine with graphics hardware that is incapable of supporting such features, the effects are unavailable, since software emulation is usually prohibitively slow.

The graphical effects that the engine currently provides to “The Meadow” virtual environment include:

3.1.1 High Dynamic Range (HDR) Lighting

Traditionally, lighting calculations are performed using just 8 bits per colour channel, resulting in a finite set of brightness values. Reality, however, is quite different; for example, the sun at noon is around 100 million times brighter than starlight [Reinhard et al. 2006]. This is not achievable using traditional techniques, and requires a much higher range of values for accurate representation. Such a higher dynamic range of values can now be simulated on modern graphics hardware by using higher precision floating point numbers for the colour channels. HDR lighting has become an increasingly popular effect in computer games, and as such is included as an integral part of the programmable pipeline in our engine.

Current computer displays, with a very few (costly) exceptions, are still limited to a low dynamic range of colours that can be represented. Therefore, any simulation that performs calculations in a high dynamic range must convert them in order to be successfully displayed. This process is known as tone mapping, and often includes a number of parameters that can be adjusted. In our system, a virtual ‘exposure’ value is presented, mimicking that of a real camera’s – the principles of which should be familiar to our students. This exposure can either be adjusted manually, or an automatic system can be specified at the program configuration level. The automatic system is based on that used in the game Half Life 2 [Mitchell et al. 2006], and involves building up a basic histogram of the scene luminance, by each frame drawing pixels within a different luminance range to an off-screen buffer, and asynchronously querying the results.

Another popular HDR effect implemented in the graphics engine is that of the bloom, or glare. Here, overly bright parts of a scene appear to glow very slightly. This is achieved by extracting the brightest parts of the rendered frame, blurring them, and then compositing the result with the original.

3.1.2 Image Processing Techniques

Continuing with the camera analogy, a depth of field effect is provided, along with an approximation to motion blur. The depth of field, which results in an area that is in focus, with objects too close or too far away being blurred, is implemented in a similar manner to that described by Gillham [2007]. The depth of field effect relies on accessing the depth buffer and evaluates how blurred a pixel should be, based on its depth difference from the ‘in focus’ area – the centre of the screen. This level is then used to interpolate between the original image and a blurred version. Motion blur is approximated simply by blending the current frame with all the previous ones, the weighting of the current frame determining the amount of blur. Both effects are adjustable.

Two further visual effects are provided as post-processing techniques. The first is saturation, which allows for adjusting between full colour and grey scale; the second is vignette, which displays a darkening around the edges of the screen.

3.1.3 Object Shaders

A number of modern effects are provided within object shaders. The most notable of these is the ground shader, which displays either a grass texture or cobble stones. At a given point on the ground, the surface type is determined by a low resolution texture map that is stretched to fit the playable area within “The Meadow”. To allow for smooth transitions between the two surface types, they are

both calculated within a single ground shader and the results are blended accordingly. The grass surface is relatively simple, and relies on a grass texture as its foundation, calculating flat diffuse lighting due to the sun, lightning, and the ‘artificial’ light generated from a lamp post object. In contrast to the grass, the cobble stone surface is fairly complex. Using a height map and a normal map, it provides parallax mapping [Welsh 2004] and normal mapping. These give the illusion of a greater level of geometric detail than is actually present. During wet weather conditions, the cobble stones change their reflective properties, and a reflection approximation based on the ambient background colour is applied in low parts of the height map to simulate pools of water collecting in the gaps between stones (see figure 3).

3.1.4 Weather System

A sky system with dynamic weather is included, with basic user adjustable controls: a haze value, and a generic weather value that varies the cloud cover. The effect involves a dome for the sky colour, and a separate dome for the clouds. While the sky colour was originally calculated using the relatively complex method by Preetham et al. [1999], following Heigl’s implementation guidelines [Heigl 2004], its high computational cost prompted the adoption of an empirical solution using data collected from photographic sources. The result approximates the sky using a series of simple colour blends which are computationally cheap and, although basic, provide surprisingly good results.

The cloud system is applied to a cloud-dome, which is slightly smaller than the sky dome. To create the illusion of perspective, the texture coordinates are adjusted within the shaders. An approximation based on the process described by Elias was taken for the cloud simulation process [Elias 1998]. This approach involves the continual gradual linear interpolation between pre-generated two dimensional textures of Perlin noise [Perlin 1985]. The generic weather value is passed into the shader, and is used to determine the cloud coverage which can be changed from clear to overcast to thunderstorm. The shader lights the cloud layer by taking into account the sun direction and a single lightning source. Lightning is generated when the weather value reaches its upper quarter, with the chance of a strike gradually increasing. The lightning is an approximation of high-level inter-cloud lightning and does not display visible streaks, although this is planned as a future extension. The generator only generates one strike at a time, the parameters of which are passed into various shaders, such as those for the cloud layer and the ground.

One of the most important aspects of bad weather is precipitation. The weather values ranging in the upper half are used to determine rain intensity. Rain is shown as a dynamic texture in front of the camera, and also through small rain splashes on the ground (see figure 3). The techniques for both are inspired by those used by Tatarchuk [2006], where the rain splashes are small billboards – polygons aligned to face the camera. An animated splash texture is then applied to these.

3.2 Scripting Module

A recent trend in computer games is to make the games extensible. The method by which this extensibility is most often realised is by the use of more or less complex scripting systems. Scripting removes a large part of the – previously hard-coded – internal game logic from the game engine and transforms it into a game asset. Robert Huebner’s case study on the use of scripting languages in computer games details how this is often achieved [Huebner 1997]. Our engine uses a Lua scripting interface [Jerusalemshy et al. 1996]. The scripting language Lua is currently the language of

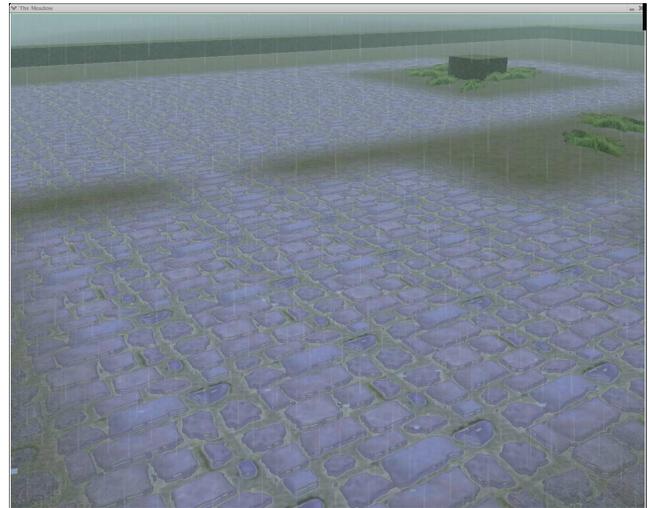


Figure 3: Rain on “The Meadow”.

choice for building the scripting solutions in many computer games. It is a generic programming language that was originally designed to be used to extend programs by adding various scriptable features, which is why the creators of Lua have dubbed it an “extensible extension language”. Since its first conception, Lua has been used to add scripting to a number of bestselling computer games. We use Lua primarily for scene initialisation, i.e. scene graph setup, with node operations provided as functions to the scripted interface, thus, it is from within the scripting environment that the scene graph hierarchy is constructed. The engine also uses Lua scripts for determining responses to user input. The GUI management system provides Lua functions, which allow for setup and the registration of Lua functions as callbacks to handle user interaction with the GUI.

3.3 Virtual Machine Module

The virtual machine used in the engine is a kind of parallel stack-based machine, allowing the creation of several simultaneously running processes, each of which has its own stack to keep different programs separate. It uses pre-emptive multi-tasking with round-robin scheduling to execute loaded processes during its run cycle. An improvement on the ZBL/0 virtual machine [Anderson 2004], it retains full backwards compatibility to bytecode compiled from programs created using the ZBL/0 core language. In the case of ZBL/0 the programming language was designed specifically as an educational tool with the definition of NPC (Non Player Character) behaviour in First Person Shooter (FPS) games in mind [Zerbst et al. 2003], which is reflected in the functions and procedures of the language, only a few of which remain present in the C-Sheep language. The instruction set of the virtual machine includes pointers as well as facilities for the creation of aggregate data types (arrays and record structures). While much of this is unused as it exceeds the requirements of the C-Sheep C language subset, the provision of this rich feature set provides upwards compatibility to possible future revisions of the system, as well as the possibility of source language independence. This can easily be demonstrated with ZBL/0 programs that can be executed within the virtual machine, in which case the NPC controlled by the ZBL/0 program in “The Meadow” is the sheep entity.

To execute native C-Sheep programs in “The Meadow” they must first be compiled with an external compiler which translates from

the C-Sheep subset of the C programming language to the virtual machine bytecode. This is where our system differs from the more integrated development environments of other systems. While this slightly complicates the use of our system, it does create greater flexibility by freeing “The Meadow” from being bound to the C-Sheep language. This potentially makes it targetable by compilers for different languages, i.e. a Java based J-Sheep or Pascal based P-Sheep could be created with relatively little effort.

3.3.1 C-Sheep Language

The C-Sheep subset of the ANSI C programming language [Anderson and McLoughlin 2006] was designed in accordance with some of the introductory language design principles proposed by McIver and Conway [1996], of which we especially considered the provision of a set of non-overlapping language features as important. C-Sheep implements the C control structures that are required for teaching the basic computer science principles encountered in structured programming, these being the (unconditional) sequence, conditional statements and loops [Böhm and Jacopini 1966]. Its instructions allow users to control the actions performed by the sheep actor and to query changes in the virtual world (e.g. the current state of the weather).

A feature in which C-Sheep differs from other mini-language based systems, that aim to provide an environment with minimal complexity, is that C-Sheep allows the declaration and use of variables, which some might consider as a potential problem. There are, however, other problems with variable free languages that can occur at the moment of transition to a real-world system, as identified by Untch [1990]. By including variables, the migration to a real-world language can be delayed and the transition will be smoother.

4 C-Sheep in the Classroom

Our system was designed as a teaching tool for the first term of a first year computer programming unit in a computer animation degree, originally planned for deployment in the 2006/2007 academic year. Unfortunately this target could not be met due to mainly hardware constraints that made it impossible to run the finished prototype of “The Meadow” in the labs assigned to the first year cohort, but we are planning to use our system in this role from the start of the next academic year.

The system itself, however, lends itself to many other uses beyond the simple introduction to computer programming with C.

4.0.2 Graphics Programming

In graphics programming, a prominent theme in the second year programming unit of our course, we have used the “The Meadow” to illustrate the usage of OpenGL for writing graphics programs. This is an area of use which we intend to explore further, as “The Meadow” provides us with a visual aid and test platform for demonstrating a wide range of techniques, from the simple usage of OpenGL, as we have already done, to the use of shaders and specific graphical effects.

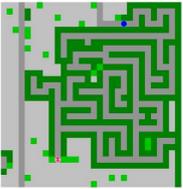
4.0.3 Languages & Compilers

Another theme in the second year computer programming unit are formal languages, parsers and compilers. Here we have used the simple structure of the C-Sheep C language subset as a sample language in classroom exercises. As “The Meadow” can be used with other languages than C-Sheep, it could also be used as the target for a compiler for a simple model language like Oberon-0 [Wirth 1996].

Exercise: Solving Problems the Sheepish Way



“Haggis” the sheep has a problem:
He is at the entrance of a maze (43/11).
His favourite ball however is at an exit on the opposite end of the maze.



1. Formulate a strategy for “Haggis” to find his way through the maze and develop an iterative algorithm that uses this strategy.
2. Derive a C-Sheep program from that algorithm and test the program either using “The Meadow” or the companion library
3. Modify the above program from an iterative to a recursive solution - use a function “go” which attempts to take the next step for the sheep and which calls itself (*recursively*) until the ball is found.
4. Expand the above program so that “Haggis” the sheep returns to the entrance of the maze after he finds his ball. This should be done as an addition to the recursive solution.



Figure 4: A programming lab assignment.

4.1 C-Sheep Companion Library

The C-Sheep system includes a counterpart library for programs written in the ANSI C programming language. The functions provided by this library mirror the C-Sheep instructions for the virtual entities in the virtual machine. As suggested by Untch, the purpose of such a library is to simplify the migration from the educational mini-language to real-world systems [Untch 1990], in our case from C-Sheep to the C programming language. Using the library, C-Sheep programs can be compiled into an executable using a normal off-the-shelf C/C++ compiler, allowing novice programmers to make an easy transition from using the C-Sheep system to C. The compiled executable can then be run from within the native working environment of the operating system.

The library allows advanced students to tackle more complex problems than those that can be solved within “The Meadow”, such as the use of dynamic data structures like linked lists and trees. The library is currently available to students in a stable and usable state, however lacking on the visual side (not yet as pretty as we would like as currently only visualising C-Sheep programs in a 2D top-down view).

4.2 Classroom Experience – a Simple Exercise

Helped by the fact that C-Sheep uses a reduced, but not minimal syntax, including data types, we have used the C-Sheep system in the context of algorithm design exercises in the second year programming unit of our course. After being introduced to both the C-Sheep system consisting of the C-Sheep compiler and “The Meadow”, as well as the C-Sheep companion library for use with a normal compiler, students were presented with an algorithm design exercise: the design and implementation of a simple maze solver (see figure 4). This is a kind of exercise that could easily be used in the first year computer programming unit as it provides a problem that can be used for the teaching of iteration and recursion.

A question that students frequently ask when confronted with similar toy-problems in programming exercises is the relevance of that problem to real-world problems or - in extreme cases - to their exams. In this case we encountered fewer such questions, which we believe to be due to the visual nature of the problem. This made it easier for the students to recognise situations in which the solution to this problem might be useful.

4.3 Observations & Student Comments

When trying the system in the classroom we first expected to encounter comments similar to those reported by Pierre et al [Pierre et al. 2006]. They had first tried to teach using the 2D top-down Robocode [Li 2002] system, but found students disliked the 2D view as they were used to 3D views from the games they were playing. We therefore anticipated our students to dislike the 2D counterpart library and to favour “The Meadow” 3D virtual environment. Instead, to our surprise, we found that while our less experienced students did prefer the 3D view, some of our students actually preferred the primitive 2D top-down graphics of the counterpart library. The students who preferred the 2D library were those who had either a greater interest in computer programming or some prior programming experience, and while they liked the look of “The Meadow” they thought it distracted from the main task, i.e. the solving of problems. While this could only be incidental, it suggests that additional research is necessary to determine if the choice of dimension for the graphical representation is a relevant factor when it comes to motivating students with different levels of competence.

When asked, the overall reaction was that C-Sheep “makes programming more interesting”. The consensus was that the system provided “a fun way to program” or that using it was just “fun”. Some of the students had previously been exposed to the Logo turtle and therefore recognised its similarity to the instructions available in C-Sheep. This led them to compare it to the turtle, with a few even suggesting we offer the alternative choice of a turtle to complement the sheep. The use of the sheep as the programmable entity - which was a random pick (or rather a joke) - itself turned out to be a good choice: we found that the use of a cartoon style sheep was attractive to both male and female students, the former considered it to be “cool”, whereas the latter thought of the sheep as “cute”. This finding, although encouraging, must be interpreted with care as the number of students participating in the exercise was relatively small, making this anecdotal evidence which cannot be deemed representative. However, we believe this calls for further investigation, as research suggests that gender inclusiveness at an early point in computer science education is a contributing factor to success in the field, as female students usually have “significantly less pre-college programming experience” [Murphy et al. 2006]. This makes it especially important to motivate female students to practice programming so they do not fall behind their male counterparts.

5 Summary & Future Work

We have presented C-Sheep, a system to simplify the teaching of computer science principles using the C programming language by means of a mini-language subset within a state-of the art 3D computer game-like virtual environment. For easy migration from our system to real world compilers, the system is complemented with a counterpart library for use with real-world compilers. One of the enhancements that we intend to make is to extend the C counterpart library to use better graphics. Coupled with this improved visual representation, the control structures, operators and aggregate data types (arrays and record structures) accessible by any real-world C compiler could make an expansion of the C-Sheep C language subset unnecessary. Instead of “The Meadow”, this enhanced counterpart library could be used to facilitate the introduction of these elements of C language syntax, providing a more seamless transition to the real-world compiler. A major issue with the current implementation of the C-Sheep system that requires additional attention is the support of a wider range of graphics hardware, as the prototype implementation of the underlying games engine has some

steep graphics hardware requirements. The provision of additional shaders for lower-spec hardware and vendor independence is only a matter of course and will not only simplify the deployment in our computer labs, but also make the support of additional platforms a possibility. During use in the lab, students discovered a number of potential problems, e.g. slight incompatibilities between C-Sheep compilers on different platforms (Linux and Windows) which we hope to fix before the full deployment of our system. Students also made suggestions for improvement that we will address in the future. Their feedback leads us to believe that further work on the C-Sheep function library, i.e. the instructions available for the control of the sheep, is required to enhance interactivity with the virtual environment in “The Meadow”. Another issue raised by students was the available documentation, mainly consisting of a description and explanation of the available functions, which many of them considered to be insufficient. Poiker explains how novice programmers write programs employing a mixture of “copy and paste” with “trial and error”, requiring many examples and good documentation [Poiker 2002]. We therefore intend to provide the next version of our system with a more detailed manual and comprehensive examples to simplify the introduction of the system into the first year programming unit in the next academic year. Finally we are planning to include a JIT (Just In Time) compiler for C-Sheep into “The Meadow” to allow C-Sheep program development and testing within the confines of the virtual environment.

Acknowledgements

First and foremost we would like to express our gratitude towards our supervisor, Prof. Peter Comminos. Without his support and encouragement this project would never have reached its current state. We also have to mention our collaborator on this project, Steffen Engel, as well as our other colleagues for their comments and suggestions that have contributed to this project. Finally, a word of thanks must go to our students for giving us useful feedback.

References

- ANDERSON, E. F., AND MCLOUGHLIN, L. 2006. Do robots dream of virtual sheep: Rediscovering the karel the robot paradigm for the plug&play generation. In *Proceedings of the Fourth Game Design and Technology Workshop and Conference (GDTW 2006)*, 92–96.
- ANDERSON, E. F. 2004. A npc behaviour definition system for use by programmers and designers. In *Proceedings of CGAIDE 2004*, 203–207.
- BEAUBOUF, T., AND MASON, J. 2005. Why the high attrition rate for computer science students: some thoughts and observations. *ACM SIGCSE Bulletin* 37, 2, 103–106.
- BIERRE, K. J., AND PHELPS, A. M. 2004. The use of muppets in an introductory java programming course. In *CITC5 '04: Proceedings of the 5th conference on Information technology education*, 122–127.
- BIERRE, K., VENTURA, P., PHELPS, A., AND EGERT, C. 2006. Motivating oop by blowing things up: an exercise in cooperation and competition in an introductory java programming course. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, 354–358.
- BÖHM, C., AND JACOPINI, G. 1966. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM* 9, 5, 366–371.

- BOSELLI, L., 2004. Gun-tactyx - historical background. <http://gameprog.it/hosted/guntactyx/info.php#intro0>.
- BRUSILOVSKY, P., CALABRESE, E., HVORECKY, J., KOUCHNIRENKO, A., AND MILLER, P. 1997. Mini-languages: A way to learn programming principles. *Education and Information Technologies* 2, 1, 65–83.
- CARTER, L. 2006. Why students with an apparent aptitude for computer science don't choose to major in computer science. *ACM SIGCSE Bulletin* 38, 1, 27–31.
- COOPER, S., DANN, W., AND PAUSCH, R. 2000. Alice: A 3-d tool for introductory programming concepts. *Journal of Computing Sciences in Colleges* 15, 5, 107–116.
- DANN, W., COOPER, S., AND PAUSCH, R. 2000. Making the connection: Programming with animated small world. In *Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, 41–44.
- ELIAS, H., 1998. Cloud cover. <http://freespace.virgin.net/hugo.elias/models/m.clouds.htm>.
- GHOSH, P., 2006. Computer industry 'faces crisis'. BBC News Online: <http://news.bbc.co.uk/1/hi/technology/6155998.stm>.
- GILLHAM, D. 2007. Real-time depth-of-field implemented with a postprocessing-only technique. In *ShaderX5: Advanced Rendering Techniques*. Charles River Media, 163–175.
- HARTNESS, K. 2004. Robocode: using games to teach artificial intelligence. *J. Comput. Small Coll.* 19, 4, 287–291.
- HEIGL, S., 2004. Simulating nature. <http://www.eisscholle.de/index.php?d=devel/openmountains&sub=daylight>.
- HUEBNER, R. 1997. Adding languages to game engines. *Game Developer* 4, 9.
- IERUSALEMSCHY, R., DE FIGUEIREDO, L. H., AND CELES, W. 1996. Lua - an extensible extension language. *Software: Practice & Experience* 26, 6, 635–652.
- JENKINS, T. 2001. The motivation of students of programming. In *ITiCSE '01: Proceedings of the 6th annual conference on Innovation and technology in computer science education*, 53–56.
- KELLEHER, C., AND PAUSCH, R. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys* 37, 2, 83–137.
- KELLEHER, C. 2006. Alice: Using 3d gaming technology to draw students into computer science. In *Proceedings of the Fourth Game Design and Technology Workshop and Conference (GDTW 2006)*, 16–20.
- KERNINGHAM, B. W., AND RITCHIE, D. M. 1988. *The C Programming Language*. Prentice Hall.
- KOCH, J., AND MOHR, J. 2006. Gute Fächer, schlechte Fächer. *Der Spiegel*, 50, 64–79.
- LAHTINEN, E., ALA-MUTKA, K., AND JÄRVINEN, H. 2005. A study of the difficulties of novice programmers. *ACM SIGCSE Bulletin* 37, 3, 14–18.
- LATINGA, S., 2004. Simple directmedia layer introduction. <http://www.libsdl.org/intro.php>.
- LI, S., 2002. Rock 'em, sock 'em robocode! IBM developerWorks: Java technology - <http://www-106.ibm.com/developerworks/library/j-robocode/>.
- MARK, W. R., GLANVILLE, R. S., AKELEY, K., AND KILGARD, M. J. 2003. Cg: a system for programming graphics hardware in a c-like language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, 896–907.
- MCIVER, L., AND CONWAY, D. 1996. Seven deadly sins of introductory programming language design. In *Proceedings of Software Engineering: Education and Practice (SE:E&P'96)*, 309–316.
- MCLOUGHLIN, L., AND ANDERSON, E. F., 2006. I see sheep: A practical application of game rendering techniques for computer science education. Poster at Future Play '06 Conference.
- MITCHELL, J., MCTAGGART, G., AND GREEN, C. 2006. Shading in valve's source engine. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, 129–142.
- MOSKAL, B., LURIE, D., AND COOPER, S. 2004. Evaluating the effectiveness of a new instructional approach. *ACM SIGCSE Bulletin* 36, 1, 75–79.
- MURPHY, L., RICHARDS, B., MCCAULEY, R., MORRISON, B. B., WESTBROOK, S., AND FOSSUM, T. 2006. Women catch up: gender differences in learning programming concepts. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, 17–21.
- PAPERT, S. 1980. *Mindstorms: Children, Computers and Powerful Ideas*. Basic Books.
- PATTIS, R. E. 1981. *Karel the Robot, a Gentle Introduction to the Art of Programming*. John Wiley and Sons.
- PERLIN, K. 1985. An image synthesizer. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques SIGGRAPH '85.*, 287–296.
- PHELPS, A. M., AND PARKS, D. M. 2004. Fun and games: Multi-language development. *ACM Queue* 1, 10, 46–56.
- PHELPS, A. M., J., B. K., AND PARKS, D. M. 2003. Muppets: multi-user programming pedagogy for enhancing traditional study. In *CITC4 '03: Proceedings of the 4th conference on Information technology curriculum*, 100–105.
- POIKER, F. 2002. Creating scripting languages for nonprogrammers. In *AI Game Programming Wisdom*. Charles River Media, 520–529.
- PREETHAM, A. J., SHIRLEY, P., AND SMITS, B. 1999. A practical analytic model for daylight. In *Proceedings of the 26th Annual Conference on ComputerGraphics and Interactive Techniques.*, 91–100.
- REINHARD, E., WARD, G., PATTANAIK, S., AND DEBEVEC, P. 2006. *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting*. Morgan Kaufmann.
- SHREINER, D., WOO, M., NEIDER, J., AND DAVIS, T. 2005. *OpenGL Programming Guide*, 5th ed. Addison-Wesley.
- SMITH, R., 2003. The open dynamics engine. <http://www.ode.org>.
- TATARCHUK, N. 2006. Artist-directable real-time rain rendering in city environments. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, 23–64.

- UNTCH, R. H., 1990. Teaching programming using the karel the robot paradigm realized with a conventional language. On-line at: <http://www.mtsu.edu/~untch/karel/karel90.pdf>.
- WELSH, T. 2004. Parallax mapping. In *ShaderX3: Advanced Rendering With DirectX And OpenGL*. Charles River Media, 89–96.
- WIRTH, N. 1996. *Compiler Construction*. Addison-Wesley.
- ZERBST, S., DÜVEL, O., AND ANDERSON, E. 2003. *3D-Spieleprogrammierung*. Markt + Technik.