Designing a computer graphics course for first year undergraduates

Neil A. Dodgson & Andrew Chalmers

Victoria University of Wellington, New Zealand

Abstract

We document the challenge of designing a technical computer graphics course for undergraduate students who have taken only a single undergraduate programming course and have not yet had to take any mathematics beyond high school. This course is the introduction to a major in computer graphics within a Bachelor of Science degree. We needed an introduction to the rigour of computer graphics that would attract students to continue with the major, that would provide a useful foundation for that major, and that could be attempted with minimal prerequisites.

1. Introduction

In the 2016 academic year, Victoria University of Wellington introduced a major in Computer Graphics into its Bachelor of Science degree. The structure of the major is shown in Figure 1. One feature of the major is that students are able to take the first-year introductory computer graphics paper (CGRA151) having taken only a single introductory programming course and with no guarantee that they have taken any university-level mathematics. The challenge was to design a course that was accessible to such students.

The traditional approach to undergraduate computer graphics teaching is to have an introductory course at third or fourth year level within a computer science major [Ohl86]. This gives plenty of scope for students to acquire necessary programming and mathematical skills prior to meeting computer graphics. As part of the new major, Victoria University had decided that is was vital that there was at least one first-year course explicitly in computer graphics, in order to attract students to the major. Further, this course should ideally be accessible to students from outside the major, who may have yet to take any university-level mathematics.

We discuss the motivation for the design of the overall major (Section 2) before delving into the detail of how we constructed our first-year course in computer graphics, covering choice of programming language (Section 4), syllabus (Section 5), assessment (Section 6). We follow this with our reflections on the tutors' experience (Section 7) and lessons learnt from the first year of running the course (Section 8).

2. Design of a major in computer graphics

The major was introduced with advice from and in collaboration with the New Zealand visual effects and computer games industry. Our aim is to produce graduates who can be employed within those industries, with a higher level of skills useful to the industry than would be achieved from a straight computer science degree.

We are aware of the research that shows that it is unhelpful to graduates to design a degree that produces a jack-of-all-trades, someone who knows a little about all aspects of games or effects but is master of none of them [LH11]. These industries need both amazing artists and brilliant programmers, each of whom should ideally have some understanding of and sympathy for the other. We therefore focussed on producing, within the Bachelor of Science, a major that would produce graduates who were immersed in the technical programming skills, but we also included a couple of courses from the University's design degree, to give graduates some understanding of how those on the artistic side are trained and how they are taught to think.

The first year, as designed, contains four strands: programming, mathematics, graphics, and freely chosen options. In the graphics strand is, first, the School of Design's standard introductory course on using Maya, followed by the new introduction to computer graphics, described in detail in this paper. The Maya course has no prerequisites; the new computer graphics course, taught by the School of Engineering and Computer Science, requires only a single programming course as prerequisite. In addition to the majority constituency of science students, the course is also open to engineering students who want a taste of computer graphics and to design students who want to experience a more rigorous programming course; this allows those on the artistic side of the discipline an opportunity to see how the technical students are trained.

3. Overview of the first-year course

The course, as taught in 2016, ran over twelve teaching weeks. It comprised 34 lectures and 12 laboratory sessions, with laboratories alternating between tutorial sessions (in odd weeks) and marking

DOI: 10.2312/eged.20171020

Year 1 Term 1	DSDN132 Intro. to Maya	COMP102 Intro. to Programming	ENGR121† Intro. to Algebra	*option*
Year 1 Term 2	CGRA151 Intro. to CG	COMP103 Data Structures and Algorithms	ENGR123 Logic and Probability	ENGR122 Calculus
Year 2 Term 1	MDDN241 Advanced Maya	COMP261 Algorithms and Data Structures	NWEN241 Systems Programming	*option*
Year 2 Term 2	CGRA251 Computer Graphics	*option*	*option*	*option*
Year 3 Term 1	CGRA350 Real-time 3D CG	COMP307* Artificial Intelligence	COMP313* Game Development	*option*
Year 3 Term 2	CGRA353 Image & Video	*option*	*option*	*option*

Figure 1: *The structure of the three-year Bachelor of Science majoring in Computer Graphics. Each term contains four courses. The graphics strand is the leftmost column, with one course each term. The option courses allow students to pair the computer graphics major with further computer science courses, with mathematics or physics, or with other design courses. Notes: *COMP307 and COMP313 are recommended but can be replaced by other computer science courses. †ENGR121 is the Engineering Introduction to Algebra course; students taking the computer graphics major are required to take this mathematics course but it is not a prerequisite for CGRA151 as that course is designed to be accessible to design students and others who have not taken a university maths course.Course code abbreviations: CGRA Computer Graphics, COMP Computer Science, NWEN Network Engineering, ENGR Engineering Mathematics, DSDN Design, MDDN Media Design.*

sessions (in even weeks). The course coordinator (ND) prepared all the course materials and gave the majority of the lectures. He was assisted by another lecturer who handled two weeks of lectures and set one quarter of the final exam. We recruited a team of eight tutors, from amongst postgraduate and advanced undergraduate students. One of those tutors (AC) assessed all of the assignments prior to their release to the students and helped coordinate the team of tutors.

The course had five marked programming assignments (worth 30% of the final mark), a mid-term test (10%) and a final examination (60%). This emphasis on the final examination is typical of our first year courses. Courses in later years tend to have more focus on assignment work.

In the first year of giving the course there were 163 students, with an 82% pass rate. Of those 163, twelve (7%) failed to engage fully with the course, either by not sitting the exam or by submitting fewer than three of the five assignments. The students were drawn from the Faculties of Science (79 with 76% pass rate), Engineering (78, 88% pass rate), Design (4, 50% pass rate), and Commerce (2, 100% pass rate). A further 66 students pre-registered for the course (41 science, 10 engineering, 8 design, 7 others) but withdraw, mostly before the course started but a few in the first two weeks of the course. Of the 163 students registered on the course, student records show that 97 (60%) had previously taken a university-level algebra course.

4. Choice of programming language

Students coming onto the new course take introductory programming through one of two routes. Most were expected to come in through the computer science route (in either Science or Engineering), having taken an introductory course in *Java* (COMP102). A minority were expected to come in from Design, having taking a creative coding course taught in *Processing*. We had the choice of using one of these or of introducing a new language. We discussed using *Python* or *C++*, as both of these are needed later in the major and both are important in the visual effects and post-production industry. A compelling case can be made for teaching Python [Jen04] and it has become the most popular introductory teaching language at top US universities [Guo14]. However, it was thought inappropriate to introduce a dramatically different language to students who were only beginning to consolidate their programming ability. It was especially inappropriate when the purpose of the course was to teach computer graphics concepts rather than programming per se. Similar arguments apply to *C++*, though the complexity of that language meant that it was never a serious contender.

The choice thus came down to teaching with *Java*, using an in-house graphics library, or with *Processing*. *Processing* [RF15] is a programming language explicitly designed to be accessible to and usable by artists and designers. Reas and Fry created it "...to teach the fundamentals of computer programming within a visual context, to serve as a software sketchbook, and to be used as a production tool." [RF14]. Schweiter et al. chose *Processing* for their introductory computer graphics course precisely because it is well-suited to people with minimal programming experience and as a rapid prototyping tool [SBG10]. For our purposes it is important that *Processing* is primarily aimed at producing visual output and has an excellent in-built graphics library. It has a nicely designed development environment and it is trivial to produce visual results within the first five minutes of using the environment. It is based

on *Java* and almost everything a student knows about *Java* can be transferred directly to *Processing*. We therefore settled on *Processing*, despite misgivings from some staff that the majority of students would be having to handle *Processing* simultaneously with learning more advanced *Java* programming on the concurrent Data Structures and Algorithms course. The strong similarities between the two languages allowed us to argue that the students would largely be consolidating their programming skills and that this outweighed any confusion over the minor differences between the languages.

5. Syllabus

Students were permitted to take this introductory course without having taken any university mathematics course. We therefore depended on high school mathematics and we knew that there would be a wide variation in mathematical ability in the class. This had a dramatic effect on the syllabus that could be taught. When we consider other, similar courses, we find that rather more mathematical ability is assumed. For example, Paquette assumes at least basic skills in calculus, linear algebra and trigonometry [Paq05], while Shesh, who explicitly discusses the problems of students who are struggling with mathematics, assumes students will have taken at least two university-level calculus courses and a discrete mathematics course [She15].

For the course content, we drew heavily on material already available from the course-coordinator's previous second-year computer graphics taught at the University of Cambridge. This had been originally designed in the 1990s around Foley et al's seminal text [FvDFH90], with updates added year-on-year. It was itself influenced by the previous course at Cambridge, taught by the late Neil Wiseman, which had its roots in the 1980s. However, both of these earlier courses assumed significant undergraduate-level mathematics prerequisites, and so we had to be selective in what was re-used, what was modified, and what was completely new.

We faced an interesting challenge. We wanted the course to contain rigorous material that would form the foundation for teaching advanced algorithms in future years, which meant that we could not use a textbook aimed at artists and designers, such as that by Reas and Fry [RF14]. On the other hand, the graphics textbooks aimed at computer science and engineering majors tend to assume considerable mathematical prerequisites.

We chose to use one of the standard texts as a basis for the course: Shirley and Marschner's *Fundamentals of Computer Graphics* [SM09]. A quick glance through this book will reveal that, as expected, much of what is taught depends heavily on mathematics. Therefore our introductory graphics course of necessity includes a mathematics primer and is selective in which parts of the text it references. It was also necessary to use a *Processing* text [RF15] and to provide high-quality lecture material comprising copies of slides with substantial accompanying notes. Both text books were made available electronically through the University Library, meaning that students had to spend no extra money on hard copy resources. We were unable to secure electronic rights to the most recent (fourth) edition of Shirley and Marschner, so the students had access to the earlier (third) edition, which is equally useful for a course of this nature.

The course was structured around four main themes:

Programming in *Processing*. Here we drew on the Reas and Fry's texts [RF14, RF15] to provide a solid introduction to using a 2D graphics library. Because all of the assignments were to require programming, this part of the course needed to be taught first. We used it as a refresher to reinforce what students had previously learnt in their introductory programming course. With regard to programming, our aim was that they should apply what they already knew rather than constantly having to learn new control structures and data structures.

Fundamental algorithms. In addition to teaching student how to use a particular graphics library, we wanted to teach some of the simpler algorithms that form the foundation of computer graphics. We wanted students to understand the fundamental ways in which a graphics card converts a library call into drawing individual pixels. We chose a number of straightforward short algorithms that demonstrate the range of operations required, and also the types of optimisations that can be made to these algorithms. The algorithms were: line drawing (DDA, Bresenham, Midpoint), circle drawing (Midpoint), Bézier spline drawing, triangle filling, polygon filling, line clipping, and polygon clipping. One important consideration in the choice of algorithms was to show how the same ideas recur in different contexts. For example, the line drawing algorithm is used in Bézier spline drawing, is modified to make a circle drawing algorithm, and forms part of a triangle filling algorithm. This part of the course thus forms a series of practical examples in algorithm design. All of these algorithms are presented in 2D. Paquette comments: "In every introductory CG course, many topics such as curves and transformations are first introduced in 2D and then extended to 3D... [but] bringing the same topic from 2D to 3D is not a trivial extension" [Paq05]. We believe it is justifiable to stick to 2D, given that the students are in their first year and will have further courses in which to explore 3D computer graphics.

Underlying mathematics. In addition to algorithms, we needed to lay foundations for the more advanced mathematics in later courses. We needed to teach basic vector and matrix algebra, which was a refresher for most but not all students. We used these to show how transforms are implemented in homogeneous coordinates in 2D and 3D, tying this into the an explanation of how *Processing* internally implements its various transformation operations. We also had to teach algebra required by some of the algorithms: in particular, how to find the closest distance from a point to a line, and how to find the intersection point of two lines. These again allowed us to demonstrate how algorithms are designed and how implementations can be optimised.

Human vision and display technology. The final quarter of the course had to be material that was not required on any of the practical programming assignments, as the final assignment started before this material was lectured. It was a relatively straightforward decision to use this part of the course to teach on human vision and display technologies: a range of topics that are accessible to anyone with a high school education. We covered the human visual system, the fundamental limitations of human vision, its implications for the design of displays, a range of display technologies including LCD, DMD and printers, and the various colour spaces used in computer graphics.

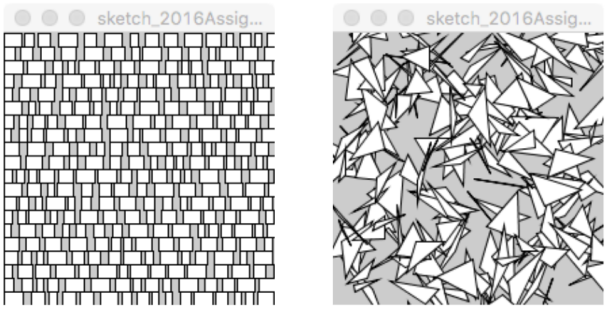

Figure 2: *Two of the examples of controlled randomness that students had to reproduce for Assignment 1 Completion.*

6. Programming assignments

The course had five programming assignments, worth a total of 30%. These offer models for others to refashion for other courses.

The first four of the five assignments all followed Victoria University's conventional model for computer science assignments, where an assignment has three parts:

Core worth 65%. This is work that is accessible to a C-grade student. Successful completion gives at least a pass mark.

Completion worth 25%. This completes the work undertaken in the Core. It is accessible to a B-grade student but will be a stretch for a C-grade student.

Challenge worth 10%. This takes students well beyond the Completion part, consolidating what has been learnt in a new challenge. It should be completable by A-grade students but may be a stretch even for them.

This structure acknowledges the wide range of programming expertise in any given class. It allows the less able students to achieve a pass mark in an assignment while giving the most able something challenging to get their teeth into.

On the new computer graphics course, each assignment was accompanied by a work sheet. The work sheet took the student through a range of small *Processing* exercises, which prepared the student for the assignment. Students had a laboratory session every week. In odd weeks, the tutors were available to help with the work sheet and the assignment. In even weeks, tutors marked students' work in one-on-one sessions allowing direct feedback and also allowing them to check that the student understood their submission. The first four assignments, briefly, covered the following.

1. Introduction to *Processing*. The Core and Completion were based on exercises in the texts by Reas and Fry [RF14, RF15]. The accompanying work sheet introduced some concepts related to randomness in computer graphics, and the Completion exercises consolidated this by requiring students to produce output with carefully controlled randomness (Figure 2). The Challenge in this assignment was to create a piece of 'art'. The idea here was to allow students to exercise freedom of expression rather than being limited to following a set of instructions slavishly. At this point, students had had only three lectures and were still getting to grips with *Processing*. Nevertheless, some interesting ideas were investigated. A video of some of the best is available on Vimeo: https://vimeo.com/195207746

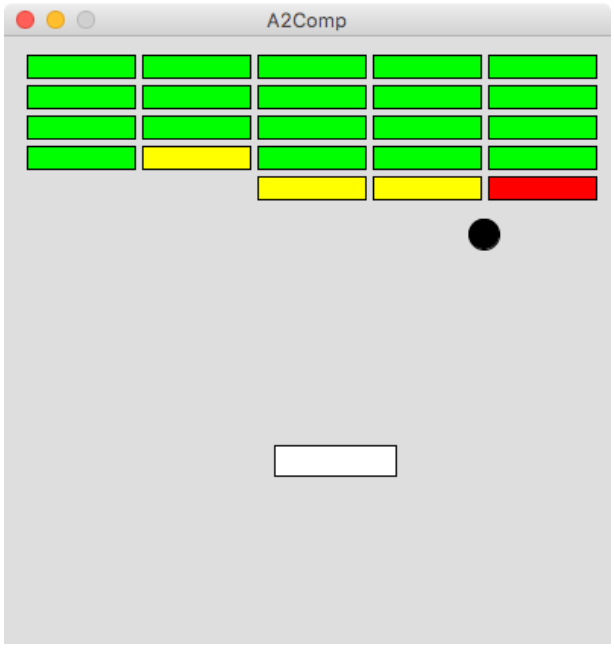

Figure 3: *One student's solution to Assignment 2 Completion.*

2. Basic physical simulation. This assignment was based on BreakOut, the simple bat and ball game (Figure 3). The Core exercise was to implement a 2D bouncing ball (a circle) that bounced off the sides of the window and off a movable rectangular bat controlled by the mouse. The accompanying worksheet prepared students for this by leading them through creating a ball that moved horizontally and bounced off the left and right side of the window. Completion was to add a series of rectangular targets that changed colour after each hit and would eventually (after sufficient hits) vanish. Challenge was either to make a bat of arbitrary orientation with correct bouncing, or to use multiple balls, bouncing correctly off one another. The majority of students found that the Challenge was beyond them. In addition, we discovered that, while it is straightforward to bounce a ball off a static object, it is challenging to get the ball to collide correctly with a moving bat and most students were unable to get this interaction to work correctly.

3. Curves and transforms. The aim of this exercise was to get students comfortable with handling cubic Bézier and Catmull-Rom splines, both of which are built in to *Processing*, and to be able to handle transforms. The idea was that the student would create a series of $C1$-continuous spline curves that would form a 'roadway' (Figure 4) and then a shape (a 'car') would move along this spline path, keeping itself correctly oriented at all points. The accompanying work sheet showed how to do this for Bézier splines. The assignment required students to implement the same idea for Catmull-Rom splines. We found that the work sheet had explained the mechanism in such detail that there was little challenge in converting from Bézier to Catmull-Rom, and that the intention to teach the use of transforms was compromised by the students being able

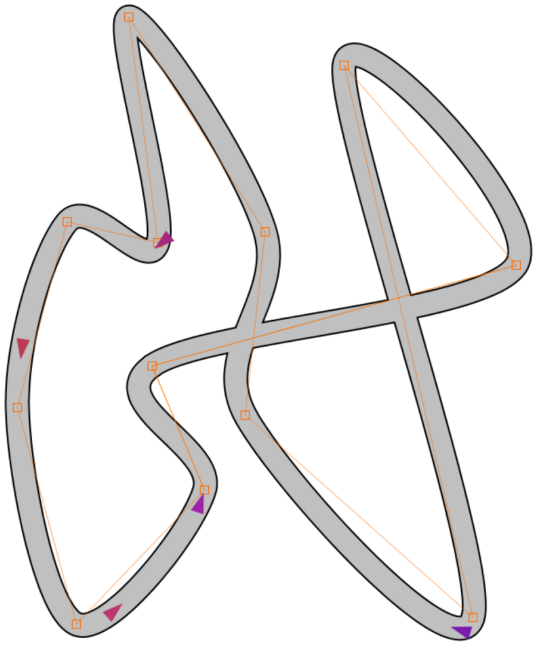

Figure 4: *An exemplar for Assignment 3: the orange squares are the control points of a series of Catmull-Rom cubic splines, which students were required to allow to be moved using the mouse. The spline itself drawn as a broad grey stroke. on top of a slightly broader black stroke, giving the look of an edge to the stroke. Five 'cars' are represented by oriented triangles.*

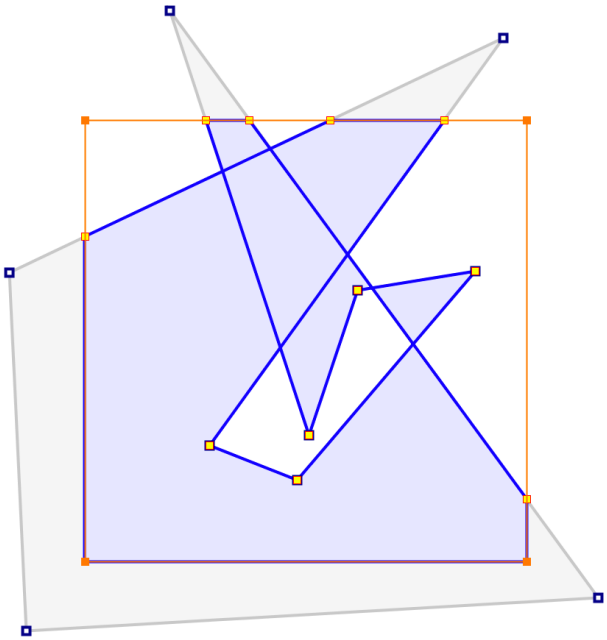

Figure 5: *An exemplar for Assignment 4. The solid orange squares are the control points of the clipping polygon. The heavy-bordered squares are the vertices of the polygon to be clipped. All of these can be moved by the user. The clipped polygon is outlined and shaded in blue. The clipped parts are in grey.*

to copy and paste code without needing to understand how the transforms worked. The Challenge was to prevent the 'cars' from hitting one another. Few students attempted this and it is extremely challenging to get collision detection to work in all situations.

4. Polygon clipping algorithm. This was the only assignment in which students attempted to implement one of the fundamental computer graphics algorithms. We asked the students to implement the Sutherland-Hodgman reentrant polygon clipping algorithm [SH74] [FvDFH90, §3.14]. The Core was to implement clipping against a single infinite edge. Completion was to complete the reentrant algorithm. A further requirement was to implement a user interface so that the user could move any polygon vertex on either the clipping or clipped polygon (Figure 5). The Sutherland-Hodgman only clips against a convex polygon, so the Challenge was to implement an algorithm that clipped any two arbitrary polygons against one another, regardless of convexity. The assignment was designed to test learning of the algorithms and ability to implement the necessary mathematics (finding the intersection of two straight lines). We were disappointed at the number of students who found this difficult, which indicates that we need to spend more of the course on the basic mathematics. One mistake we made here was that we did not provide a work sheet for this assignment, trusting by this point that students would be able to handle an assignment without needing a lead-in. We were wrong. On the other hand, we had a small number of students who made a good attempt at the Challenge, which required exploration of algorithms that were not taught in the course.

6.1. Capstone project

The final assignment was designed to finish the course with a capstone project that gave the students considerable creative freedom. It was also designed to ease students from first year to second year ways of thinking. In first year, all assignments are small, deadlines are weekly (or every two weeks), and they are highly structured. In second year, assignments are larger, take longer, and require better time-management as it becomes impossible to complete an assignment successfully by starting the night before the deadline.

The final assignment was to create a 2D game or 'interactive art work' of the student's own design. Students were told about this on Day 1 of the course, to give lots of time to think about possibilities. They were given one week to write and hand in a one-page plan, which was discussed with a tutor before they started programming. They were then given three weeks in which to do the programming. Many students made initial forays into programming before getting formal approval, which is to be encouraged because part of good planning is to test out potential ideas.

As part of the one page plan they were required to commit to deliverables at the end of each of the three weeks of programming. Our recommendation was that, by the end of Week 1, all of the critical components of the project worked, at least roughly, and that,

in extremis, the student could hand this in. By the end of Week 2, the student should have a reasonable submission that would get a decent mark. Week 3 should be spent polishing the submission to produce something that would score an excellent mark. Of course, we actually expected most students to find that it takes longer than they plan to get things working, so many students needed the third week to give time to fix things that went wrong. We did not monitor whether they produced their deliverables in the timeframe they suggested; we simply assessed the final product.

Most students enjoyed this assignment, enjoyed the freedom to create something of their own, and produced good results. Students were asked to submit a one-page reflection on their experience. These were fascinating documents. A frequent comment was that the work had taken longer than expected and that the time-consuming parts of the project were not where they had expected them to be. For example, one student chose to make all of her assets (loadable image files) using Adobe Illustrator and discovered that this took her as much time as all of the programming combined. These lessons about time management are good to learn early in your university career. The team of tutors were impressed by the quality and range of outputs, with almost all students scoring over 70%. A video of some of the best is available on Vimeo: https://vimeo.com/196225486

7. Tutors' experiences

Students were assigned to attend one tutorial session per week in which they could interact with the tutors, though they could attend other sessions if there were spaces available. We had ten sessions in each week, with students being able to self-allocate to their preferred session using an online system. Tutorials were all held in the same computer room. We selected a small computer room with 20 workstations to provide an intimate teaching space. We initially restricted each tutorial to 16 students to provide workstations for up to four tutors, though we had to relax this restriction when late registrations took our numbers close to, and then above, 160 students.

In each session, there were at least two tutors. In odd weeks, the two tutors were available to help with the work sheet and the assignment. In even weeks, tutors marked students' work in one-on-one sessions allowing direct feedback and also allowing them to check that the student understood their submission. Because the one-on-one sessions required more time, there were four tutors in even weeks, to handle up to 20 students. Each one-on-one discussion was up to 10 minutes long. Tutors found that it was beneficial for the student to go through the assignment and explain what they had done. For time efficiency, tutors would direct the conversation to key points of interest, such as important concepts or objectives of the assignment, rather than discussing particular bugs in their code.

The tutors were briefed by the course coordinator each week. This involved a group discussion between the course coordinator and the tutors on the key objectives of the assignments, allowing immediate feedback from the tutors to the coordinator on how the students were feeling from the previous assignment, and also allowing for modifications to the upcoming assignment, such as difficulty scaling. In even weeks the briefing session involved all tutors marking the same small number of student assignments, to allow debugging of the marking scheme and to ensure that tutors were applying the same criteria in marking. Students were rotated round the marking tutors in their laboratory so that they saw a different marker for most assignments. This ameliorated any bias caused by a particular marker being 'tougher' or 'easier' on students.

Tutors received assignments one week beforehand to allow them to attempt the assignment themselves. The tutors would share information of what they had learnt with the other tutors. This gave them useful insight into the common difficulties that could arise from the students.

The odd-week sessions, in which the students attempted worksheets, proved useful. The work sheets were designed to be relatively straightforward and walked the student through some of the key concepts of the assignment. It was particularly useful in getting the student to think about the problem at hand before attempting the solve the problem themselves. The tutors' experience was that some of those who skipped the worksheet had more trouble completing the core of the assignment and tended to ask preliminary questions of the assignment, whereas students who completed the worksheet tended to get onto the completion sooner and would have questions about the advanced key objectives of the assignment.

The even-week one-on-one marking sessions allowed students to explain how they went about solving the problems in the assignment and where they had issues. This allowed the tutors to understand where most students had the most difficulty. This information proved useful for the following odd-week's session, allowing the tutor to go around the class and provoke conversations on topics that the students seemed to have the most trouble with. This ranged from programming questions, such as describing object-oriented programming and class structures, to mathematical questions, such as the use of a dot product.

8. Lessons learnt

We found that students were, largely, competent at programming in *Processing*, with almost all students submitting passing attempts at all five assignments. The capstone projects demonstrated that the majority of students had a good grasp of programming and reasonable design ability. The initial worries that students would get confused between *Java* and *Processing* were unfounded.

With regard to the algebra that we taught, we discovered that Shirley and Marschner [SM09, Ch.6] move quickly from simple matrix algebra, accessible to our students, to eigenvalue decomposition, which was well beyond what we thought sensible to teach at this level. We therefore needed to be careful in directing students to which sub-sections of the textbook were necessary for the course and which they could safely leave until the second-year course.

Despite our attempts to teach the appropriate mathematics, there was still a large number of students that floundered in areas that a computer graphics professional finds trivial. The final exam revealed that many students had failed to grasp straightforward matrix representations of transforms. In particular, students found 3D transforms challenging. We may decide to limit the course to 2D mathematics in future years. Indeed, during the development of the course, we had decided to cut back the programming assignments

to handle only 2D computer graphics as we felt that introducing 3D for one or two assignments would be pushing the students too far. We preferred to build towards an excellent capstone project in 2D rather than a weak one in 3D. We subsequently found support for this idea in Erik Paquette's proposal for a 2D computer graphics and image processing course [Paq05], whose syllabus is similar to our own.

In order to give students a better chance with the mathematics, we need more formal support. We offered help sessions in mathematics but take up was very low, perhaps because students did not link the mathematics to the programming required, especially in assignments 2 (simple animation) and 4 (line-line intersection). In retrospect we need to have the mathematics tied more closely into the formal work sheets and assessments. Some mathematical concepts could have been better understood through programming a Processing sketch. However, few students were practising this technique. We need to highlight the idea of attempting sketches on their own, outside of the worksheet and assignments. We can encourage this in future years by providing students with appropriate supporting exercises to use in their own time,

Assignment 3 was the weakest of the five. It failed in its aim to give students insight into transforms. We are considering replacing it and perhaps combining something more rigorous on transforms in *Processing* with some assessed mathematical problems.

Some students reported a mismatch between the content being taught in lectures and what they were attempting in the assignment at the time. The theoretical content, such as 3D transformations, were considered by the students to be not useful, particularly when the students specifically desired mathematical content which was applicable to their current assignment instead.

During revision, before the final exam, students complained that there were too many algorithms to learn. We think that this indicates that many students were depending on memorising algorithms to regurgitate in an exam rather than understanding the underlying details. We will re-assess whether we are teaching too many different algorithms but we think a better solution is to work harder on showing how the different algorithms are linked so that students can construct the outline of any algorithm without having memorised the fine detail.

9. Summary

Overall, we produced a course that provides a good foundation for our second year material and we were impressed by the students' ability to program a simple game. However, we need to revisit whether a technical computer graphics course can be truly accessible to students who have not taken a university-level algebra course, and whether we therefore need to insist on an algebra pre-requisite at the expense of making the course less accessible to students from outside the Faculties of Science and Engineering.

Acknowledgements

Thanks to the anonymous reviewers for their clearly articulated recommendations that significantly improved the paper. Thanks to rest of the course team for their dedication and hard work: Dr Zohar Levi, Kieran Carnegie, Cullum Deighton, Connor Moody, Richard Roberts, Joshua Scott, Ryan Sumner, and Anneka Wijetunge.

References

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, second edition, 1990. 3, 5

[Guo14] Philip Guo. Python is now the most popular introductory teaching language at top U.S. universities. BLOG@CACM, 2014. http://cacm.acm.org/blogs/blog-cacm/176450. 2

[Jen04] Tony Jenkins. The first language: A case for python? *ITALICS: Innovation in Teaching and Learning in Information and Computer Sciences*, 3(2):1–9, 2004. 2

[LH11] Ian Livingstone and Alex Hope. Next Gen: Transforming the UK into the world's leading talent hub for the video games and visual effects industries. Technical report, NESTA, 2011. http://www.nesta.org.uk/publications/next-gen. 1

[Ohl86] Mark R. Ohlson. The role and position of graphics in computer science education. In *ACM SIGCSE Bulletin*, volume 18, pages 232–237, 1986. 1

[Paq05] Eric Paquette. Computer graphics education in different curricula: analysis and proposal for courses. *Computers & Graphics*, 29(2):245–255, 2005. 3, 7

[RF14] Casey Reas and Ben Fry. *Processing: A programming handbook for visual designers and artists*. MIT Press, second edition, 2014. 2, 3, 4

[RF15] Casey Reas and Ben Fry. *Getting Started with Processing*. Maker Media, second edition, 2015. 2, 3, 4

[SBG10] Dino Schweitzer, Jeff Boleng, and Paul Graham. Teaching introductory computer graphics with the Processing language. *Journal of Computing Sciences in Colleges*, 26(2):73–79, 2010. 2

[SH74] Ivan E. Sutherland and Gary W. Hodgman. Reentrant polygon clipping. *Commun. ACM*, 17(1):32–42, January 1974. 5

[She15] Amit Shesh. Teaching graphics to students struggling in math: An experience. In *Eurographics (Education Papers)*, pages 23–29, 2015. 3

[SM09] Peter Shirley and Steve Marschner. *Fundamentals of Computer Graphics*. CRC Press, third edition, 2009. 3, 6