



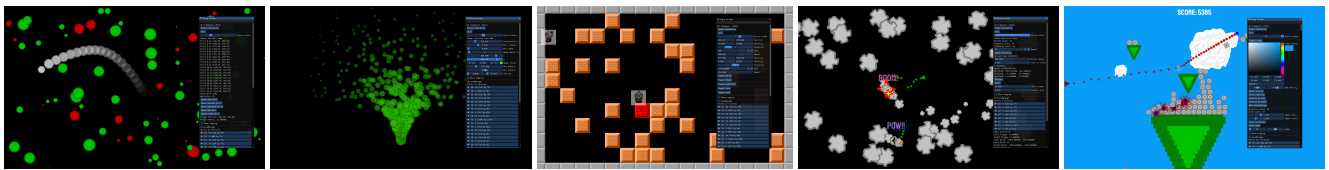


# Teaching Game Programming in an Upper-level Computing Course Through the Development of a C++ Framework and Middleware

S. Hooper<sup>1</sup> , B. C. Wünsche<sup>1</sup> , P. Denny<sup>1</sup> , and A. Luxton-Reilly<sup>1</sup> 

<sup>1</sup>University of Auckland, New Zealand



**Figure 1:** Five example formative exercise scenes — each with real-time debugging features. From Left: (a) Alpha blending trails with collision detection and response, (b) The ParticleEffectsEditorScene, (c) Data-driven level loading, (d) The AsteroidsCloneScene with static text rendering and win/lose conditions, and (e) The PhysicsGameScene with static and dynamic rigid bodies, projectiles, and particle effects

## Abstract

The game development industry has a programming skills shortage, with industry surveys often ranking game programming as the top skill-in-demand across small, mid-sized, and large triple-A (AAA) game studios. C++ programming skills are desired, however, educators can perceive C++ as too difficult to teach due to its size and complexity. We address the challenges of teaching C++ in an upper-level Game Programming course and demonstrate how learners are up-skilled in C++ game programming, providing insights and reflections on the course. We show how through careful educational-design choices, combined with scaffolding a C++ framework and contemporary middleware, it is possible to transition learners to C++ for game programming.

## CCS Concepts

• *Applied computing* → *Computer games*; • *Computing methodologies* → *Computer graphics*; • *Social and professional topics* → *Computing education*;

## 1. Introduction

There is a shortage of skilled C++ game programmers [Hau11], with game industry surveys highlighting the challenge of hiring for the specialised skills needed [Ken16; NZG22; Int23]. Previous work has discussed game programming topics and techniques to prepare learners for industry roles [IGD08; The23]. With continual advances in the game industry, including the ongoing development of *game engine* (GE) technology, changing or keeping curriculum up-to-date can be challenging [SHS\*08; Ken16]. Using C++ and middleware can lessen the challenge by helping to mitigate potential impacts on a course when changes to a GE occur. We provide insight and reflection on the techniques and technologies used at a large, urban university to transition learners to C++ for game development in an upper-level *Game Programming* course (GPC). We describe a reusable C++ framework alongside a variety of contemporary middleware and associated techniques for

game programming, rather than using a specific GE. Focusing on the core principles of game programming and transferable knowledge of a widely used programming language such as C++, can positively engage learners, encouraging them to not only develop their game programming skills and knowledge, but also make them more versatile as game programmers.

## 2. Related work

Historically, game programming courses tended to take a C++-based approach [CRG05; PKR06; AK07], sometimes with Microsoft's *DirectX* application programming interface (API) [RE07; PNS\*08]. In the late 2000s many courses used now-defunct technology, such as *Microsoft XNA*, a C#-based API [ZLS08; MP09]. Since the mid-2010s, two commercial GEs have been widely used in teaching: *Unity* (<https://unity.com/>) and *Unreal Engine* (<https://unrealengine.com/>) [DBEK17], but other en-

gines have also been used [Rit09; ELGG10; SP23]. In an educational setting, the choice of GE is not trivial [FC20] and using a GE can hide the underlying complexity of game development [TN22]. There have even been courses at postgraduate level focusing on GE development itself [AG16]. Some game programming courses have taken the approach to integrate open source libraries [AK07], and ANDERSON and PETERS provided insights into possible APIs to use in Computer Graphics (CG) education, alongside points to consider when using middleware in an educational setting [AP10]. Educators do need to be aware of the burden that can be introduced by using multiple APIs within their courses [Wil01]. Learning C++ alongside learning CG programming with an API can be challenging [WCS\*18; UKW22]. Some educators also believe the language is too large and complex to teach, even with a multiple course sequence in a three-year degree programme [FC20].

### 3. The Game Programming course (GPC)

The GPC was a third-year computing elective delivered across a twelve-week semester, with two 2-hour directed contact sessions per week. Learners were also expected to spend around six to eight hours per week on self-directed activities for the course. After an initial introduction to C++, learners implemented key *GP Framework* (GPF) functionality through formative exercises, while also upskilling in game programming techniques. A combination of C++ scaffold and middleware was used to craft the reusable framework. As per RODEN and ETHEREDGE, game programming concepts can be introduced with a 2D approach, and then evolved into the equivalent 3D counterpart [RE07]. The GPC used this strategy.

#### 3.1. Middleware used — GPF integration

As well as supporting framework development, the use of contemporary middleware exposed learners to technologies used in commercial game development. Learners were often familiar with these technologies having seen splash screens at the start of games they had played. Getting to see the developer side of how middleware worked and could be integrated into the GPF, as well as their APIs, documentation, and starter sample code, was not only exciting for learners, but also a practice that should be used in advanced technical courses [Wil01; PNS\*08]. Middleware was updated each semester with new versions integrated and tested.

The initial API chosen for graphics rendering in the GPF was *OpenGL* (<https://www.opengl.org>), as learners were already familiar with this from the prerequisite CG course. To build a framework to support cross-platform development, *Simple Direct-Media Layer* (SDL) (<https://www.libsdl.org>) was selected. SDL is used in commercial games. It provides low-level access to graphics hardware — it can be integrated with graphics APIs — and game input devices. Additional libraries can extend its functionality, such as *SDL2-image* ([https://wiki.libsdl.org/SDL2\\_image](https://wiki.libsdl.org/SDL2_image)) for texture loading, and *SDL2\_ttf* ([https://wiki.libsdl.org/SDL2\\_ttf](https://wiki.libsdl.org/SDL2_ttf)) to rasterise TrueType fonts — learners supplied their own free fonts. The *OpenGL Extension Wrangler Library* (GLEW) (<https://glew.sourceforge.net>), a cross-platform open-source C/C++ extension loading library, was also integrated. *Dear ImGui* (<https://www.dearimgui.com>) is a graphical user interface library, distributed as `.cpp` and `.h` files that

can be added to a C++ project. Helper code for different graphics APIs and platforms is provided, including but not limited to, *SDL2* and *OpenGL*, as well as code examples demonstrating usage of the API. *Dear ImGui* was added to the GPF to enable the creation of interactive visual debugging interfaces. *FMOD* (<https://www.fmod.com>) was used for audio, enabling the loading and playing of sound effects, streaming of music files, and generation of audio effects. It has a C++ API, good documentation, and pre-made code examples. The physics engine *Box2D* (<https://box2d.org>) was added for collision detection and response. This middleware has many pre-made physics samples and learners were able to use these to upskill. The *Lua* (<https://www.lua.org>) programming language was used to enable scripted data-driven design within the framework. The *RakNet* (<https://www.jenkinssoftware.com>) API was used for networking. Additionally, other middleware was integrated depending on game design needs, for example, Microsoft's *DirectX* and the *DirectX Tool Kit* (<https://github.com/microsoft/DirectXSDK>) with its C++ helper classes, was sometimes used as an alternative CG API, and Nvidia's *PhysX* (<https://developer.nvidia.com/physx-sdk>) was used for 3D physics simulation. Learners were incrementally introduced to each piece of middleware with formative exercises that worked to integrate the middleware into the GPF.

#### 3.2. Getting started — Initial code skeletons

To introduce the GPF, an initial interactive lecture session reviewed the skeleton source code of the main classes and discussed framework design choices, highlighting relationships between classes and the steps to get the framework up-and-running. Each class provided the opportunity to contextualise different programming ideas in C++ alongside game programming techniques.

To begin implementing the GPF, a new *Visual Studio* (<https://visualstudio.microsoft.com/vs/>) Solution and Project was created and configured to use the *SDL2* and *GLEW* libraries (configuring access to `.h` includes and `.lib` linker settings). The *Debug* and *Release* build targets were configured for the 32-bit x86 platform, each differentiated by outputting a uniquely named `.exe` to enable testing of each target's game executable. For some learners, this was their first encounter with the implications of Debug and Release builds. The project was configured per target to output to separate directories for intermediate build data, and the *Build Output* directory was configured to be the *Working* directory where all game-ready assets (for example, texture, game configuration, and level data files) were stored and loaded from.

With learners having spent prior courses programming with Java and C, there were many C++ topics to introduce. This included, but was not limited to, pointers, references, class declaration (`.h` files), class definition (`.cpp` files), destructors, copy constructors, operator overloading, and the C++ tool chain (preprocessor, compiler, linker). The difference between using a `#include` versus forward declaration, and using `char*` strings contrasted with the `std::string` are the sorts of C++ aspects that are unfamiliar to learners, yet important to becoming a productive C++ game programmer. C++ concepts were carefully introduced alongside the development of the GPF, incrementally expanding the complexity of C++ used through the reveal of the framework.

The GPF established good practices for code modularity and reusability by highlighting the difference between class declaration (.h) and definition files (.cpp). Each GPF .h followed a common template format of methods first, followed by properties, each ordered with access modifiers as follows: `public`, `protected`, `private`, as well as preprocessor header guards. Object lifespan and data life cycles in C++ at runtime can be surprising to learners, and one technique to help highlight this behaviour is to make the `copy constructor` and `assignment operator` `private` within a class declaration. This will cause the C++ compiler to generate an error at compile time if the learner accidentally causes an object copy to occur — helping to highlight the difference between working with a concrete instance of an object, versus an object reference via pointer — all different and potentially surprising behaviour when compared to the learner’s prior Java and C experiences.

Formative exercises often used `TODO` comment stubs to guide learners through developing new features, for example, a comment to add memory leak detection code. As classes were introduced, different opportunities to discuss common game programming patterns arose. The `Logger` introduced the Singleton and this was a useful discussion point [GS08], and a chance to check learner understanding of the `static` keyword.

The program’s entry point was light-weight, with a simple `main` function that initialised the `Game` instance, and then ran the game loop — an essential game pattern [GS08]. Next, the `Game` class was implemented — again using the Singleton pattern, the appropriateness debate of this pattern could be quickly revisited with learners [GS08]. The `Game` class’s initialisation was responsible for loading required game resources, and controlling the game loop — the calls to the `Process` and `Draw` methods.

The `Scene` interface was critical to establishing different in-game scenes. To create a new scene, learners derived a subclass realisation that implemented unique behaviour for loading, processing, and drawing, as well as containing game entities that belonged to the scene. The design of this interface exposed learners to unfamiliar C++ features, such as using the `virtual` keyword to create `abstract` and `pure virtual` methods, the importance and behaviour of a `virtual destructor`, and by using the debugger, the creation of the v-table at runtime. Also, the `DebugDraw` abstract method could be used by subclasses to implement `Dear ImGui` debug visualisations. Many exercises were built by deriving a new `Scene` subclass, creating a contained area for a particular game scenario, technical feature experiment, or a test sandbox, such as `AlphaBallsScene`, `ParticleEffectsEditorScene`, `BombermanCloneScene`, `AsteroidsCloneScene`, or `PhysicsGameScene` (see Figure 1).

The `Renderer` interface was also critical, as this class provided an abstraction for drawing using the graphics card. In the early years of the course, the initial implementation used the `SDL2` 2D rendering functionality, with the option for learners to then extend it to support 3D using `OpenGL` or `DirectX`. Learners could use either the fixed-function or programmable shader pipeline, depending on their comfort level, interest level, and desire to be challenged. The 2022 `Renderer` design used the programmable shader pipeline, with `OpenGL` and `OpenGL Shading Language` (GLSL), but still kept the interface simple to encapsulate shader complexity. The `Renderer.cpp` implementation

wrapped `SDL2` based upon the `SDL Wiki` examples ([https://wiki.libsdl.org/SDL2/SDL\\_CreateRenderer](https://wiki.libsdl.org/SDL2/SDL_CreateRenderer)), demonstrating the Facade pattern [GS08]. The `Renderer` also provided an opportunity to discuss the usage of `const` in C++, and further examples of when to use `#include` versus forward declaration, and the difference between `pass-by-value` and `pass-by-reference`.

The `Renderer` demonstrated another useful pattern, a Factory which created renderable `Sprite` objects. Learners were unfamiliar with graphical shader programming, as the prerequisite CG course only covered the fixed-function pipeline. Simple GLSL shaders, using the old version 3.3 as a starting point, were used to draw sprites, enabling learners to see how these shaders are used to control rendering and allowing them to become familiar with how shader code is written, compiled and then loaded at runtime within the framework. Following this, further introductory graphical effects, such as blur, were developed using shaders. Additional shaders were developed as the course progressed and these were used to create different graphical special effects and rendering styles, including additional post-processing effects, as well as 3D transformation and animation techniques. Alongside this a `TextureManager` was developed to support loading and retrieving textures, avoiding duplicating the same resource in memory. This initial design became the blueprint for other resource managers, including audio files, static rasterised text, and 3D meshes. This also formed the basis of an initial in-class discussion about the creation of a C++ generic template class that could be used for any sort of resource management within the framework.

From here, further framework modules were developed, including an input system using `SDL2` that supported game controllers, keyboard and mouse, as well as helper functions for random number generation, a particle effects system, and game-based data structures, such as object pools and quadrees — altogether creating a robust starting point for the creation of a game prototype. Initial GPF source code can be retrieved this repository: <https://github.com/SteffanHooperUoA/InitialGPF>

### 3.3. Validation of C++ skills with game industry expectations

To encourage learners’ reflection on the C++ skills they had learnt, they were invited to try the *King Pro Challenge* mobile application (<https://apps.apple.com/app/king-pro-challenge/id692742660>) to test their abilities against industry expectations, and to join the *Forage Electronic Arts Software Engineering* job simulation (<https://www.theforage.com/simulations/electronic-arts/software-engineering-awbf>).

## 4. Reflections and conclusion

One concern each year was how learners would react to the focus on C++ programming, and whether they were more attracted to the immediacy of a contemporary GE versus the satisfaction of crafting their own GPF. The interactions throughout showed that learners were indeed interested in building their own C++ framework and developing it was possible with third-year learners. The GPF helped the course achieve its aim of providing practical foundational knowledge using contemporary tools and middleware.

Using middleware and C++ helped to reduce or localise issues

and the possibility of technical debt impacting the course, and the potential impact of major GE changes was mitigated. The variety of middleware exposed learners to different APIs, which they were quickly able to work with. It was enough to be able to use each middleware for a particular purpose, with complete API mastery left for later. Further middleware could be integrated to the GPF based upon game project needs. Learners could also transition to different GEs after completing the course without being afraid to experiment, and feeling they had much greater insight into how commercial engines work.

Introducing *Dear ImGui* visualisations, alongside data-driven design techniques, empowered learners to quickly iterate on design choices. Gameplay elements could be tweaked in real-time while the game was running, and game and framework code could be driven with dynamic data, allowing level generation and ease of refining game design. Having learners see the benefits of exposing game design behaviours to the immediate mode graphical interface, where the game design features could be tested or tuned in real-time at runtime, helped them realise that not having to recompile a project to adjust design choices was empowering.

Centrally-based university learner surveys were anonymously conducted at the end of semester with learners asked to comment on the best aspects of the course — some examples of such comments were as follows: “*Learning C++ in a crash course*”, “*Learning game dev and C++ at the same time was a fun challenge*”, and “*There was a great learning curve in using C++ with creating games in this paper which I value greatly and appreciate*”.

## Acknowledgements

We thank *Auckland University of Technology* for offering the GPC — *COMP710 Game Programming* — and all the GPC students.

## References

- [AG16] AMADOR, G. and GOMES, A. “A Video Games Technologies Course: Teaching, Learning, and Research”. The Eurographics Association, 2016. DOI: [10.2312/eged.20161027](https://doi.org/10.2312/eged.20161027) 2.
- [AK07] AMRESH, A. and KARNICK, P. “Creating Interest in Computer Graphics by Teaching Game Development”. The Eurographics Association, 2007. DOI: [10.2312/eged.20071011](https://doi.org/10.2312/eged.20071011) 1, 2.
- [AP10] ANDERSON, E. F. and PETERS, C. E. “No More Reinventing the Virtual Wheel: Middleware for Use in Computer Games and Interactive Computer Graphics Education”. The Eurographics Association, 2010. DOI: [10.2312/eged.20101013](https://doi.org/10.2312/eged.20101013) 2.
- [CRG05] COLEMAN, R., ROEBKE, S., and GRAYSON, L. “Gedi: A Game Engine for Teaching Videogame Design and Programming”. *J. Comput. Sci. Coll.* 21.2 (Dec. 2005), 72–82. ISSN: 1937-4771 1.
- [DBEK17] DICKSON, P. E., BLOCK, J. E., ECHEVARRIA, G. N., and KEENAN, K. C. “An Experience-Based Comparison of Unity and Unreal for a Stand-Alone 3D Game Development Course”. Bologna, Italy: ACM, 2017. DOI: [10.1145/3059009.3059013](https://doi.org/10.1145/3059009.3059013) 1.
- [ELGG10] ESTEY, A., LONG, J., GOOCH, B., and GOOCH, A. A. “Investigating Studio-Based Learning in a Course on Game Design”. *Proc. of the 5th Int. Conf. on the Foundations of Digital Games*. Monterey, California: ACM, 2010, 64–71. DOI: [10.1145/1822348.1822357](https://doi.org/10.1145/1822348.1822357) 2.
- [FC20] FACHADA, N. and CÓDICES, N. “Top-down Design of a CS Curriculum for a Computer Games BA”. *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. Trondheim, Norway: ACM, 2020, 300–306. DOI: [10.1145/3341525.3387378](https://doi.org/10.1145/3341525.3387378) 2.
- [GS08] GESTWICKI, P. and SUN, F. “Teaching Design Patterns Through Computer Game Development”. *J. Educ. Resour. Comput.* 8.1 (Mar. 2008). ISSN: 1531-4278. DOI: [10.1145/1348713.1348715](https://doi.org/10.1145/1348713.1348715) 3.
- [Hau11] HAUKKA, S. “Working in Australia’s Digital Games Industry: Consolidation Report”. *ARC Centre of Excellence for Creative Industries and Innovation and Queensland University of Technology in partnership with the Games Developers’ Association of Australia* (2011) 1.
- [IGD08] IGDA. *IGDA Curriculum Framework*. Available at <https://web.archive.org/web/20090419214800/http://www.igda.org/wiki/images/e/ee/Igda2008cf.pdf>. 2008 1.
- [Int23] INTERACTIVE GAMES & ENTERTAINMENT ASSOCIATION. *Australian Game Development Survey FY 2023 Report*. Available at <https://igea.net/wp-content/uploads/2023/12/AGDS-2023-Report-Final.pdf>. 2023 1.
- [Ken16] KENWRIGHT, B. “Holistic Game Development Curriculum”. *SIGGRAPH ASIA 2016 Symposium on Education*. SA ’16. Macau: ACM, 2016. DOI: [10.1145/2993352.2993354](https://doi.org/10.1145/2993352.2993354) 1.
- [MP09] MORRISON, B. B. and PRESTON, J. A. “Engagement: Gaming throughout the Curriculum”. *Proceedings of the 40th ACM Technical Symposium on CS Education*. SIGCSE ’09. Chattanooga, TN, USA: ACM, 2009, 342–346. DOI: [10.1145/1508865.1508990](https://doi.org/10.1145/1508865.1508990) 1.
- [NZG22] NZGDA. *NZ Interactive Media Industry Survey 2022*. <https://nzgda.com/news/nz-interactive-media-industry-survey-2022/>. [Accessed 22-Jan-2024]. Nov. 2022 1.
- [PKR06] PARBERRY, I., KAZEMZADEH, M. B., and RODEN, T. “The Art and Science of Game Programming”. *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*. Houston, Texas, USA: ACM, 2006, 510–514. DOI: [10.1145/1121341.1121500](https://doi.org/10.1145/1121341.1121500) 1.
- [PNS\*08] PARBERRY, I., NUNN, J., SCHEINBERG, J., et al. “SAGE: A Simple Academic Game Engine (Extended Abstract)”. (Jan. 2008) 1, 2.
- [RE07] RODEN, T. E. and ETHEREDGE, J. “Educating game programmers”. *Message from the Program Committee Chair* (2007), 80 1, 2.
- [Rit09] RITZHAUPT, A. D. “Creating a Game Development Course with Limited Resources: An Evaluation Study”. *ACM Trans. Comput. Educ.* 9.1 (Mar. 2009). DOI: [10.1145/1513593.1513596](https://doi.org/10.1145/1513593.1513596) 2.
- [SHS\*08] STURTEVANT, N. R., HOOVER, H. J., SCHAEFFER, J., et al. “Multidisciplinary Students and Instructors: A Second-Year Games Course”. *SIGCSE Bull.* 40.1 (Mar. 2008), 383–387. ISSN: 0097-8418. DOI: [10.1145/1352322.1352269](https://doi.org/10.1145/1352322.1352269) 1.
- [SP23] SOBOTA, B. and PIETRIKOVÁ, E. “The Role of Game Engines in Game Development and Teaching”. Aug. 2023. DOI: [10.5772/intechopen.1002257](https://doi.org/10.5772/intechopen.1002257) 2.
- [The23] THE JOINT TASK FORCE ON COMPUTING CURRICULA (ACM) (IEEE-CS) (AAAI). *CS Curricula 2023 Version Gamma*. Available at <https://csed.acm.org/wp-content/uploads/2023/09/Version-Gamma.pdf>. 2023 1.
- [TN22] TUNNEL, R. and NORBISRATH, U. “A Survey of Estonian Video Game Industry Needs”. *Journal of Education and Learning* 11.5 (2022), 183–192 2.
- [UKW22] UNTERGUGGENBERGER, J., KERBL, B., and WIMMER, M. “The Road to Vulkan: Teaching Modern Low-Level APIs in Introductory Graphics Courses”. The Eurographics Association, 2022. DOI: [10.2312/eged.20221043](https://doi.org/10.2312/eged.20221043) 2.
- [WCS\*18] WÜNSCHE, BURKHARD C., CHEN, ZHEN, SHAW, LINDSAY, et al. “Automatic assessment of OpenGL computer graphics assignments”. *Proc. of the 23rd Conf. on Innovation and Technology in Computer Science Education (ITCSE 2018)*. New York, NY, USA: ACM, 2018, 81–86. DOI: [10.1145/3197091.3197112](https://doi.org/10.1145/3197091.3197112) 2.
- [Wil01] WILKENS, L. “A multi-api course in computer graphics”. *Journal of Computing Sciences in Colleges* 16.4 (2001), 66–73 2.
- [ZLS08] ZYDA, M., LACOUR, V., and SWAIN, C. “Operating a computer science game degree program”. *Proceedings of the 3rd international conference on Game development in computer science education*. 2008, 71–75 1.