ELECTRICAL AND INFORMATION ENGINEERING
University of the Witwatersrand, Johannesburg
Software Development II

Project 2026 — Centipede

1 Introduction

Centipede is a 2D arcade game released by Atari in 1981 [1]. It enjoyed commercial success and was subsequently ported to many different platforms. Its game mechanics were also imitated by a number of other arcade games. To understand Centipede's gameplay in more detail, you should watch it being played.

Your task for this project is to write a "Centipede"-type game for the PC. This means that your game must be based on the same game mechanics as Centipede (see Section 3.1), but you are free *and encouraged* to choose any game theme and backstory that you like.

2 Project Outcomes

On completion of this project you should be able to:

- Perform an object-oriented decomposition and analysis of a software problem involving a variety of interacting objects.
- Design and implement an object-oriented solution in C++.
- Understand and use existing software libraries in conjunction with your own code.
- Provide a test suite verifying the correctness of your software.
- Provide the requisite documentation for a software project, including an automatically generated technical reference manual.
- Work successfully in a small team to deliver a software product.

The purpose of this project is to learn software design and engineering by applying the principles and practices covered in this course to the development of a software product.

AI can be helpful for reviewing and critiquing your solution but do not use it to actually generate the solution. If you do, you will miss out on the opportunity that this project provides: the chance to hone your object-oriented modelling, coding and debugging skills. You should also be cautious when considering AI suggestions involving language features or design patterns that are more advanced than those covered in this course.

3 Game Requirements

3.1 Game Mechanics

The game's mechanics are illustrated in the YouTube video linked to above. The basic mechanics are as follows. The player controls a *bug blaster* that can move up, down, left,

and right within a small area at the bottom of the screen. The blaster can fire laser shots vertically upward.

The centipede moves horizontally as it advances downward from the top of the screen through a field of mushrooms. If the head of the centipede reaches either edge of the screen or collides with a mushroom, the centipede moves down one row and reverses direction.

If a laser shot hits a segment of the centipede, that segment turns into a mushroom. If the head is hit, the next segment in the train becomes the new head. If an interior segment, that is, a segment that is neither the head nor the tail, is hit, the centipede splits into two at the point where the mushroom is created. The first segment of the rear section becomes a new head, and the two smaller centipedes continue downward *independently*. This process repeats whenever an interior segment of a centipede is shot.

Mushrooms can be destroyed after being hit four times by the blaster. The blaster loses a life when hit by a centipede. A centipede is eliminated once all segments of the original centipede and any centipedes produced by it have been shot and turned into mushrooms.

3.2 Categorization of Features

The following subsections describe the basic functionality required for your game and list features considered minor or major enhancements. Implementing these enhancements well may earn higher marks in the *Functionality* category (see the rubric). If you want to implement a feature that is not listed, please contact me first so that I can advise you. You are welcome to include audio in your game; however, audio is not counted as a feature enhancement.

3.2.1 Basic Functionality

The following features are considered basic functionality:

- The following game objects exist: the bug blaster, a centipede with at least ten segments, and laser shots fired by the blaster.
- The blaster moves left and right in a single row at the bottom of the screen. The centipede moves horizontally as it advances downward. Whenever the centipede reaches the edge of the screen, it moves down one row.
- When a laser shot hits the centipede, the hit segment is destroyed rather than turned into a mushroom. If the hit segment is an interior segment, the centipede splits into two as described above. The segment preceding the destroyed segment becomes the new head of the rear centipede. The rear centipede then follows the centipede in front of it, forming a centipede "train".
- The game ends when either the centipede reaches the bottom row or the entire centipede train is eliminated by the blaster.

3.2.2 Minor Feature Enhancements

Minor enhancements include, but are not limited to, the following:

- There is a field of randomly placed mushrooms. If the head of the centipede collides with a mushroom, it moves down one row and reverses direction. The rest of the centipede train follows suit. *Implementing this feature is a mandatory requirement for achieving Good for functionality.*

- The game uses appropriate sprites or other detailed graphics rather than simple shapes such as rectangles, triangles, and circles.
- The blaster can move up and down, in addition to left and right, within a small area at the bottom of the screen. When the centipede train reaches the bottom row, it reverses vertical direction and moves upward through the blaster's movement area.
- The game has a scoring system with a visible display. High scores are saved between games and can be viewed.
- The blaster has more than one life, and the remaining lives are shown on the screen.
- The locations of the mushrooms are loaded from a file.

3.2.3 Major Feature Enhancements

Major enhancements include, but are not limited to, the following:

- Centipede segments that are shot turn into mushrooms. As before, a centipede that collides with a mushroom moves down one row and reverses direction. Consequently, when an interior segment is shot, the new centipede formed from the rear section immediately collides with the mushroom created by the shot segment and reverses direction. Mushrooms can be eliminated after being hit four times by the blaster. *Implementing this feature in its entirety is a mandatory requirement for achieving Excellent for functionality.*
- The game has spiders. These appear periodically and move in a random zig-zag pattern across the blaster's movement area. They may occasionally eat mushrooms. If a spider hits the blaster, the blaster loses a life. Spiders are eliminated if shot by the blaster.
- The game has scorpions. These never appear in the blaster's movement area. Scorpions move horizontally across the screen and poison every mushroom they touch; the poisoning must be shown graphically. A centipede that bumps into a poisoned mushroom dives straight down towards the blaster's movement area and returns to its normal behaviour upon reaching it.
- The game has DDT bombs. These are stationary and appear at random times and positions. A maximum of four bombs may be on the screen at any one time. A bomb explodes when shot, destroying all enemies (centipedes, spiders, and so on) and mushrooms within the blast radius.

3.3 Constraints

The game must be written in standard C++ using the `raylib-cpp` library. This library is a C++ wrapper for `raylib`. You *may not use* `raylib` *directly*. All releases must build and run on Windows, even though the libraries themselves are cross-platform.

The emphasis of this project is on *good object-oriented design* and not on fancy graphics. Good graphics are only regarded as a minor feature enhancement. With regard to the graphics, the dimensions of your game window must not exceed 1600×900 pixels.

You may not use any other libraries or frameworks that are built on top of `raylib` or `raylib-cpp`.

3.4 Pong Game Example and Starter Code

On the course website you will find the code for an example Pong game. Your game's directory structure should match that of the Pong game. To help you get started, there is a starter-code ZIP file containing the correct directory layout for your game, along with README files explaining what belongs in each directory. The project repository must contain the game source code, test source code, game assets, and project documentation.

You have been provided with a `CMakeLists.txt` file. You must not modify this file. Use CMake along with this `CMakeLists.txt` file to build your executables and documentation. We will use the exact same `CMakeLists.txt` file to build your releases for assessment.

Do not include in your repository:

- The source code for any of the libraries being used.
- Files that are produced as by-products of the build but are not necessary for running the game. Such files include project files produced by VS Code, object-code files, and so on.
- Any executables or library binaries.

4 Project Releases

Agile methodologies are a popular method of developing high quality software. They are both *iterative* and *incremental* in nature. Iterative implies that multiple passes through the phases of the software development lifecycle take place. On completion of each iteration a working build is produced that has a subset of the total functionality that is required. This is usually released to the customer in order to elicit early feedback. Each iteration results in an increment in functionality and iterations continue until the final solution is achieved. Note that your code will probably change significantly between releases. This is to be expected as your design converges on a final solution.

In order to encourage this style of development, there will be an initial check-in, two interim releases, and a final release. The deadlines are available on the course website. The two interim releases are work-in-progress releases for which a reduced set of project deliverables is required. The final release requires all the project's deliverables. The deliverables required for each release are summarized in Table 1 and described in more detail in Sections 4.1 to 4.4.

4.1 First Release: Initial Check-In

Before your group can begin work on the project, you need to complete the "check-in". One group member must download the project starter code, add their name and student number to the root-level README file, commit the change, and push the commit to the GitHub repository. The other group member must then clone the repository, add their name and student number to the root-level README file, and push the change to GitHub. Finally, the release needs to be tagged. This allows us to confirm that all group members are on board and ready to begin the project. The check-in is successful only if a commit from each group member is visible on the *main* branch, and the repo's content is correct. Each member's commit must be linked to their GitHub account.

If a group member's commit is not linked to their GitHub account, the group can still earn the check-in bonus provided that each member's most recent commit before the deadline is

correctly linked to their account. After fixing the issue, make a minor edit to the README file and commit again. If you have problems linking to your account, you may have configured your Git client with the wrong email address.

Once you have completed the check-in, you can start working on the project.

4.2 Second Release: Progression Checkpoint

For the second release, you are expected to demonstrate exploratory use of the `raylib-cpp` and `doctest` libraries. The game and test functionality need only be simple at this stage: at a minimum, you must implement the bug blaster that can move left and right. You must also provide tests verifying that it moves correctly and that its movement is restricted by the screen boundaries.

4.3 Third Release: Progression Checkpoint

For the third release, you are required to build on the functionality of the second release. You should aim to implement some minor features and, if possible, a major feature by this stage. Refer to Section 4.6 for more information on how release functionality relates to bonuses. A feature counts only if it has been implemented to a reasonable standard.

To promote writing tests concurrently with code, *basic movement tests must be provided for every moving game object* implemented by this stage, irrespective of which features you have implemented. The tests must verify conditions in which movement is allowed as well as conditions in which it is not allowed.

4.4 Final Release

Each project group must submit the deliverables listed in Table 1. In addition to the deliverables required for the earlier releases, the final release must include the following documentation:

- A brief *Architectural Overview* in `architecture.md` describing the architecture of your game at a high level. The Pong game's `architecture.md` can be used as a reference for the style, structure, and level of detail expected in this document. Describe the domain model, using diagrams where appropriate. Include an overall class diagram showing the relationships between the important classes in your game, and one or more sequence diagrams illustrating key object interactions. If a class contains a complex algorithm, explain it using a flowchart or activity diagram. Do not provide a flowchart of the overall game loop. Diagrams can be generated using Mermaid syntax, as in the Pong game's architecture document, or embedded using image files from the `architecture-images` directory.
- A *Technical Reference Manual* which should focus on describing each class's responsibilities and the signatures of all the member functions in the class's *public* interface.

All the documentation will be automatically extracted and formatted as HTML using CMake and Doxygen.

4.5 Criteria for a Correct Release

The following criteria define a *correct*, published project release on GitHub:

Deliverable	First Release	Second & Third Releases	Final Release
Tagged commit published as a GitHub release	✓	✓	✓
Accompanying release notes		✓	✓
Architectural overview			✓
Technical reference manual			✓

Table 1: Deliverables required for each release. The tagged commit from the second release onward will be built using CMake to produce a game executable, a test executable, and the documentation.

1. The commit that is tagged as the release commit must be on the *main* branch of the project repo.
2. The release tag must be named correctly. Tags must be *named exactly as follows*: `v1.0` (for the first release), `v2.0` (for the second release), `v3.0` (for the third release) and `v4.0` (for the final release). Note, the "v" is lowercase. If your tag is incorrect your release will not be found.

   You can create a trial release at any time by using a release tag other than those reserved for official releases (e.g. `v0.1`).
3. The commit tagged as the release must have been *pushed* to GitHub before the release deadline.
4. The release must have been published before the deadline.
5. *Release notes* must be published on GitHub describing, at a minimum, the game's *added*, *changed*, and *removed* functionality. The release notes must follow the Keep a Changelog format. This applies to the second release onward.
6. The released game must include a *splash screen that correctly and completely informs users of the keys used to play the game*. This applies to the second release onward.
7. The executables must build successfully using the provided `CMakeLists.txt` file and run on Windows. This applies to the second release onward.

If a release is not tagged, then the last commit on the *main* branch prior to the release deadline will be used as the release commit when assessing incremental development (Section 6.1.1) and continuous integration (Section 6.1.2).

4.6 Bonuses

A release must meet all the criteria in Section 4.5 to be considered for a bonus. To actually earn the bonus the release requirements that are described above must be achieved. Bonuses are awarded to the group, not to individual students. Bonus percentage points are added to the project mark calculated from the rubric. Note, however, that the final project mark cannot exceed 100%. Refer to Table 2 for the available bonuses.

5 Commits and Workflows

5.1 Commit Authorship and Atomicity

Each commit must be authored by one student in the group and, if feature branches are used, merged into *main* by that same student. Commits co-authored by AI tools are strictly

First Release: Initial Check-In	
Correctly completed check-in	1%
Second Release: Progression Checkpoint	
All second-release requirements met	1%
Third Release: Progression Checkpoint	
All third-release requirements met, plus basic functionality	1%
All third-release requirements met, plus basic functionality and one major feature	3%
Final Release	
Release made correctly (the rubric is used to assess the release itself)	1%

Table 2: Bonuses and bonus requirements by release.

forbidden. Commits must be linked to GitHub accounts so that we can accurately track code contributions.

Commits must be atomic in nature (watch up to 3:34). In the context of this project, the game and test source code must compile successfully, produce the required executables, and run successfully for every commit leading up to a release. The student who makes a commit is responsible for ensuring that it is atomic. This requires mutual trust within the team: it is natural to check that the code you have been working on compiles and runs, but code that you may not have been working on, such as the tests, also forms part of a commit that you make. In a high-performing team, no team member commits broken code to the repository; however, if the code cannot be trusted, then it must be checked.

5.2 Git/GitHub Workflows

Unlike the labs, there are few requirements regarding branches and merges. Suggested workflows for the project are given in the releases guide. Avoid force-pushing (overwriting history), rebasing branches, and squash-merging, as these will interfere with the assessment of your commit history and may negatively affect the group's marks. There is no need to use advanced Git techniques; use only the Git commands covered in the course.

6 Assessment

6.1 The Marking Rubric

The rubric, which is appended to this brief, indicates how the project will be assessed. Each category is rated from *Unacceptable* to *Excellent*, with each rating corresponding to a particular mark, as shown below. The overall mark is determined by averaging the category marks.

If any category receives a rating of *Unacceptable*, the final mark (overall rubric mark plus bonuses minus penalties) will be capped at 40%. In the case of jointly assessed categories

Rating	Mark
Unacceptable	0
Poor	25
Acceptable	60
Good	75
Excellent	100

Table 3: Ratings and associated marks

both students' marks will be capped. In the case of individually assessed categories only the individual will be capped. Note, however, that the final mark may be lower than 40%.

The rubric contains several categories. The following two categories require further explanation:

6.1.1 Incremental Development

Incremental development means building the system by adding small, usable pieces of functionality over time. Each increment can be integrated, tested, and evaluated before the next one is added. This makes progress visible, allows problems and design issues to be identified early (ranging from merge conflicts to incompatible features), and reduces the risk of leaving integration and testing until the end of the project. Taking small steps also means that, if a bug is detected, it is possible to return to a working state close to when the bug was introduced, making the bug much easier to find. This relies on commits being atomic (see Section 5.1) and determining the commit which introduced the bug can even be automated using `git bisect`.

To encourage this style of development, the following rule applies: any commit on the *main* branch leading up to a release, including the release commit itself, may contain no more than 150 *inserted* lines in the code files relative to the previous commit on the *main* branch. The commits analysed will be those made directly on *main* and those merged into *main*; merge commits themselves will be excluded. The number of *insertions* between two commits can be seen using:
`git diff --shortstat <earlier-commit-hash> <later-commit-hash>`. Note, if the previous release commit is not reachable from the current release commit (as in the case when the releases are on different branches) then this will automatically be taken as a violation of this rule.

Likewise, every commit on a branch leading to a tagged release must be atomic. Commits leading up to a release will be randomly sampled and must build successfully with CMake. Refer to the associated category in the rubric for details of how incremental development will be assessed.

6.1.2 Continuous Integration

Continuous integration is the practice of integrating small changes into a shared codebase frequently. It is often implemented using a service that automatically builds and tests each change to check that it works with the existing code. This helps the team identify defects and integration problems early, keeps the project in a consistently working state, and reduces the risk of discovering major problems near the end of the project.

For this project, to encourage continuously sharing code via the *main* branch rather than doing a "code dump" shortly before the deadline, the following rule applies. For any given release, every line in each source-code file will be attributed to the student who wrote or last modified it. No more than 50% of the total number of lines attributed to a student may land on the *main* branch on GitHub in the 12-hour period directly before the release commit appeared on GitHub. The landing time is when the lines *become part of the remote main branch* on GitHub, whether they are added by a direct push, by pushing a locally merged branch, or by merging a pull request. The time at which the lines were originally committed or merged locally is not considered. Refer to the associated category in the rubric for details of how this will be assessed.

6.2 Penalties

Various issues that may occur, together with the penalties that apply, are discussed below. Note that the maximum penalty for any interim submission is five percentage points, even if the submission has multiple issues.

6.2.1 Bad Commits

"Bad" commits are commits in which the work is not clearly attributable to a single person. A five-percentage-point penalty will apply when the commits leading up to a release, starting from the first commit after the previous release, contain:

- one or more commits not linked to the author's GitHub account,
- commits co-authored with an AI tool,
- commits where the author and committer are different, and
- commits where the author is not registered as a group member.

This penalty applies to all releases other than the first, giving you the opportunity to rectify issues discovered early in the project. The penalty will be applied only to the group member responsible for the bad commit or commits.

6.2.2 Lack of Teamwork

The ability to work well with others is fundamental to engineering and to software engineering in particular. This is why teamwork is explicitly one of the outcomes of this project. You may not work on your own, and project-partner changes will be considered only when a student deregisters or there is clear evidence that one group member is not contributing. You are required to act professionally and support one another in achieving the goals of the project. It is a good idea to state your expectations of one another before the project begins and to discuss how you will handle possible conflicts that could arise.

Both group members are expected to contribute roughly equally to the project. To account for substantial differences in contributions, the following rule applies. For any given release, every line in each source-code file will be attributed to the student who wrote or last modified it. The total number of lines contributed by each student will be determined using: `git summary --line *.cpp *.hpp *.c *.h`. This tool is part of git-extras, which is separate from Git and needs to be installed.

A five-percentage-point penalty will be applied for *each* of the interim releases (`v2.0` and `v3.0`) in which one group member's contribution is less than 35%. Only the group member

whose contribution is below 35% will receive the penalty. If a group member's contribution is below 35% for the final release (`v4.0`), that group member's final project mark will be capped at 40%. Note that under no circumstances may one group member commit work on behalf of the other group member.

6.2.3 Late Final Release

The School's Late Submission Penalty Policy will be strictly applied to the final deadline. Each student must read and understand the policy.

6.2.4 Plagiarism

Plagiarism-detection software may be used to compare project releases with one another and with code available on the internet. All instances of plagiarism will be dealt with severely. No two groups may submit identical or overly similar deliverables.

7 Final Thoughts

It is critical that you apply balanced effort to all aspects of the project and avoid the common trap of over-focusing on coding and functionality while neglecting other important activities such as analysis and design, testing, and documentation. Remember that implementing additional functionality will require additional effort in all other areas of the project. It is invariably better to produce a product that is acceptable in all respects than one that is excellent in only one respect and poor in all others.

Good luck and have fun!

References

[1] Wikipedia. "Centipede (video game) - Wikipedia." https://en.wikipedia.org/wiki/Centipede_(video_game), Last accessed: August 2026.

University of the Witwatersrand, Johannesburg
School of Electrical and Information Engineering
Software Development II

PROJECT ASSESSMENT FORM V0.51

	Unacceptable	Poor	Acceptable	Good	Excellent
Functionality and Documentation (game executable and Doxygen HTML output, jointly assessed)	executable does not build/run, game functionality is at the level required for second release bonus, or less	game runs but has major functional flaws OR **documentation is poor or does not exist**	all basic functionality working acceptably, documentation must be reasonable	all basic functionality working plus 3 minor features OR 1 major feature and 1 minor feature, documentation must be reasonable	all basic functionality working plus 2 major features and 3 minor features, documentation must be reasonable
C++ Design and Implementation (game source code, jointly assessed)	raylib-cpp not used, implementation violates constraints, not object-oriented, no user-defined classes, GitHub not used for version control and publishing releases	poorly chosen abstractions or many key abstractions missing, poorly designed class interfaces, inappropriate relationships between classes, patent violation of fundamental principles such as DRY, implementation more like C than C++	abstractions generally have acceptable/appropriate behaviour but some key ones may be missing, acceptable class interface design but implementation may not be well hidden, mostly acceptable class relationships, modern, idiomatic C++ mostly used	*2 out of 4* : 1) well-modelled abstractions at all levels of granularity with good interfaces which hide information 2) clean separation of presentation and logic layers 3) small classes and no long functions 4) good use of role modelling in the domain layer. No clearly wrong design decisions, modern, idiomatic C++ used	*3 out of 4* : 1) well-modelled abstractions at suitable levels of granularity, and at all layers, with good interfaces which hide information 2) clean separation of presentation and logic layers 3) small, cohesive classes and no long functions 4) good use of role modelling in the domain layer. No clearly wrong design decisions, modern, idiomatic C++ used, avoids substantial use of language features/patterns/techniques which are not covered in the course, avoids over-engineering
Automated Testing (test executable and source code, jointly assessed)	executable does not build/run, no genuine attempt at unit testing, doctest framework not used	insufficient testing - not meeting the requirement for *Acceptable,* or very limited testing due to insufficient functionality	automated test coverage of game logic is adequate and includes basic movement and collision testing for *all* game objects, some important game logic is not tested	automated test coverage of game logic includes all classes/functions responsible for the movement and collision of game objects, fair test coverage of features implemented, testing is thorough and test code is of good quality (good test names, easily understandable code, etc.)	distinguished from *Good* by one or more factors: comprehensive coverage of all features implemented and classes created; advanced use of testing framework, automated tests given for difficult-to-test functionality eg. involving randomness, gui interactions etc, avoids the use of test doubles/mocks
Incremental Development (Git history/GitHub, individually assessed)	the exact same commit error is present for all three releases (either a violation of the inserted line limit or a violation of commit atomicity)	commit errors are present for all three releases (a mix of inserted line limit and commit atomicity errors); a previously existing release tag is deleted, git history is overwritten or missing	two releases have commit errors present (in terms of the inserted line limit and/or commit atomicity)	a single release has commit errors present (in terms of the inserted line limit and/or commit atomicity)	across all releases: no commit exceeds the inserted line limit when compared to the previous commit and every commit is an atomic commit
Continuous Integration (Git history/GitHub, individually assessed)	code dump detected for the final release	code dumps detected for both interim releases but not the final release	code dump detected for the third release only	code dump detected for the second release only	no code dumps present

Notes:
All categories are equally weighted
If any category receives a rating of Unacceptable then marks are capped at 40%

Bonuses and penalties, as noted in the project brief, and the School's red book, will be added to the mark produced by this rubric to give the final mark.