

**COMP371: Computer Graphics**  
**TENTATIVE PROJECT DESCRIPTION**  
**(Revision 1.4)**

<b>COMP371</b>		<b>WINTER 2012</b>
<b>Instructor:</b>	Serguei A. Mokhov	<a href="mailto:mokhov@cse.concordia.ca">mokhov@cse.concordia.ca</a>
<b>Issued:</b>		January 16, 2012
<b>Part 1 due:</b>		Monday February 6, 2012, 13:00
<b>Part 2 due:</b>		Sunday February 19, 2012, 23:59
<b>Part 3 due:</b>		Wednesday March 14, 2012, 13:00
<b>Final project due:</b>		Tuesday April 3, 2012, 23:59

## 1 Objectives

We will make (hopefully) an open version (likely a subset) of the 1987 game “Nether Earth” originally released for a Z80-based ZX Spectrum platform, but using OpenGL and slightly modern and extendable software and visual designs. Let’s call it “Open Robot Battle Near Earth”. The original can be played online via a browser and a Java plug-in simulator at

<http://zxspectrum.net/>.

We will make an approximation of this game following the course topics and requirements bounded by time constraints of the semester. A sample remake of the game can be found here:

<http://www.braingames.getput.com/nether/>

that uses OpenGL, SDL, and C++ and was done in 2003–2004 and is supposedly cross-platform. It has a manual and screenshots and you can use source this for inspiration and an idea what you might be making in the end (with our own twist, of course). However, your own implementation must be at least 70% original comprising 30% of total borrowed code from all the sources, thoroughly documented and included in your design. Unattributed verbatim copying is not permitted, per the code of conduct and ethics. Thanks to Team 4 for providing the link.

You are encouraged to develop and release your project open-source on any hosting service and share later the link to your project with others. 5% bonus for those who actually do (e.g. using services provided by [sourceforge.net](http://sourceforge.net), [github.org](http://github.org), [code.google.com](http://code.google.com), and the like).

## 2 Groups

The project is to be done in groups 3-5 (all 4 parts). Each team is provided with a group account to coordinate their group activities and manage the project’s content (e.g. set up a CVS [1, 2] repository) to share documents, code and resources alike related to the group’s project.

## 3 Procedure

Teams sequentially complete each part of the project as “programming assignments” 1–3 that constitute deliverables followed by a complete project and a report. Parts 1–3 are to be demoed to

the lab instructors during the week after they are due. Whatever you demo, it must be identical to whatever is submitted to the EAS by the deadline (usually a midnight of the day preceding the demo week or day, see schedule for moder details). You (with the lab instructor present) or the lab instructor download the submitted code from the EAS and demo – demo need not be done the same lab days after the due date and can be done earlier if you are ready.

1. Part 1 (PA1): Modeling
2. Part 2 (PA2): Lighting and Camera Control
3. Part 3 (PA3): Texture mapping
4. Final: animation and game play, report with the software design and user manual

## 4 Deliverables

All parts are to be done in group. Each team member would be responsible for providing one or more models, components, etc. as subdivided by the team, and the team for the overall game environment.

1. 5% Part 1 (PA1)
2. 10% Part 2 (PA2)
3. 10% Part 3 (PA3)
4. 15% Final project and demo

In general, better quality works get better grades. The PA1–PA3 demos are evaluated by the implemented functionality in accordance with the requirements as well as the quality of the source code and modeling (procedural modeling, modular design of the code, well documented non-trivial code pieces in the form of comments, etc.).

### 4.1 Part 1 (PA1): Modeling.

In this part you are required to design a 3D scene that contains the initial static configuration of the game and model the following objects using the procedural modeling techniques and OpenGL primitives. Design your code such that the objects can be easily replaceable later on.

A number next to each item is a point for marking purposes. The total (45 + 2 bonus /45) is then rescaled to the PA1 weight.

- You can pick a yellow as a base color or a gray for your models; perhaps selecting distinct shades of the walls of a similar color to see the boundaries when shaded. If you have other color preferences and would like to use a richer color palette, go right ahead, so long it does not obscure the details.

It is wise to also define a utility cube of your own such that you can use to flexibly in most of your models.

You have to provide static models for:

1. (9) 1-unit robot components, per original game:
    - (1) Electronic Brain (“advanced AI”)
    - (1) Nuclear Warhead (“nuke”)
    - (1) Phasers
    - (1) Missiles
    - (1) Cannons
    - (1) Anti-gravity Chassis
    - (1) Track Chassis
    - (1) Bipod Chassis

(1) Your team number should be “engraved” on on each component. The team number should be modeled as cubes / polygons instead of textures at this stage. (It is to be textured in PA3.)
  2. (6) Terrain components.
    - (1) Basic plain floor tile. Your team number should be present on these tiles.
    - (1) Light rubble pile (can be polygons, pyramids, cones, spheres spread randomly to simulate debris)
    - (1) Medium rubble pile
    - (1) Large rubble pile
    - (2) Pits (two types of tiles: mid-section and ending)
  3. (5) “Construction” blocks, a-la the original game.
    - (1) Platform delimiters (unit-and-a-half tall, thin cylinder-on-box-on-a-box)
    - (1) Plain blocks (half-unit tall)
    - (1) Plain blocks (unit tall)
    - (1) Blocks with square hole (half-unit tall)
    - (1) Blocks with square hole (unit tall, sideways)
  4. (1) Remote control unit with the antenna, per the original (6 boxes and a cylinder, scaled and translated appropriately).
  5. (4) Model the home base per original game from the necessary blocks and with your team number instead of “H”. All this to be replaceable. A flag is a cylinder and a box (or a polygon).
  6. (5) Model component factories, as per the original game. The factories sport an upper robot component they produce. No flag required on the factories. There are 5 factories for 5 upper robot components.
  7. (5) Scene: a platform 50x50 that contains all of the above (at least one of each), spread out but fully visible.
- Use default unit size for the objects. You can scale them up and down a bit according to your aesthetic preferences.
  - (2) Place the camera 45° degrees up over the  $z$  axis backing away enough to see the whole scene.

- (2) Allow for flat and wireframe shading. Define a key for it as you see fit. Document the key. (Flat would be most visible with curved objects).
- (6) Allow definition of the map as an array. Later we can define levels and load different maps from a text file. You can refer to [3] for inspiration on this item.
- (2) Stack one or more example robots from the components as a bonus (max 47/45).

## 4.2 Part 2 (PA2): Lighting and Camera Control.

The scene still remains static in this part (no animation or game play). Marking is (36 + 7)/36. Design and develop the following:

1. (1) Allow for smooth shading (on top of the wireframe and flat shading from Part 1). Use the same key as in PA1 to toggle through the 3 display states. The key must be documented. It would help to make some parts of the blocky robots roundish more visibly.
2. (2) Model 4 platform area lights (light posts) 6 units tall, 0.30 units wide (can use cylinders and other primitives). Place them at the corners of the platform. Feel free to make reasonable aesthetic adjustments to your models of the light posts (similar in a way to street lights).

The *placing at the corners of the platform* part refers to the lights' bases being **embedded** within the walls if you have a wall there, at the outside corner of respective tiles, but its base is fully within the tile's area, i.e. not "sticking out into the void" from the platform.

3. (4) Place the four point light sources (spotlights) at the tops of the platform lights pointing down to floor such that the angles between the light direction vector  $\vec{l}$  and the platform light's core  $\vec{s}$ , as well as between the projection of  $\vec{l}$  on the floor plane  $xz - \vec{l}'$  and the corner floor boundaries  $\vec{x}$  and  $\vec{z}$  on that plane are all  $45^\circ$ . Specifically, such that  $\angle(\vec{l}, \vec{s}) = \angle(\vec{l}', \vec{x}) = \angle(\vec{l}', \vec{z}) = 45^\circ$ . As an example, this means that the "light-at" point would be  $(3\sqrt{2}, 0, 3\sqrt{2})$  when projected onto the floor plane assuming the light's position is  $(0, 6, 0)$  and the floor is centered at the origin in the  $xz$  plane.

Spotlight's effect should be easily visible when cast down onto the platform. Don't make it too dim. Let's standardize on the variable parameter of  $25^\circ$  by default for the cutoff angle.

You can model the actual "area bulbs" as a series of 3 white squares (polygons) or discs with gaps between them on a .3x.9 rectangle area. While we are not implementing an actual area light yet, but a single spotlight as a light source, it should reside at the center of the said rectangle.

4. (2) Allow lights to be turned on and off all at once and individually by designated keys. Provide and document a single key to turn off all 4 spotlights at the same time in one shot.
5. (1) Allow for ambient light to be turned on and off regardless whether the 4 other lights are on or off. Ambient light should **not** be as intense as to make spotlights indistinguishable on the platform.

6. (3) Allow each robot to have a spotlight component roughly positioned at the top of the robot's top component at the front, pointing forward-down under the 20 degrees angle between the platform and the light ray. The light source itself need not be drawn, but if you feel like you can do so. By default that light is off. This implies you have to stack at least two robots; facing whichever reasonable direction and allow keys, e.g. '1' and '2' for toggling their lights on/off.

(Note, in the future this will not scale, as one can have an arbitrary number of robots, and there are limits to number of light sources OpenGL supports, so in the future we'll only enable robot lights when we position the remote control on it and turn it on and off when we leave the robot).

7. (5) Define material properties for the models in Part 1. Make your material definitions easily replaceable. Specifically, robot components and the remote control unit should be "shiny"; rubble piles should be defuse if various color shades; main platform tiles should be diffuse gray. Select your own material properties and properly document them for the blocks, bases, flags, factories, etc.

8. (6) Allow to roll, pitch, and yaw of the camera with the arrow keys. Camera should be able to traverse the entire scene. You can re-use the CUGL [4] library if you want with proper attribution.

Assuming this is the main camera from PA1, let's fix the pitch, roll, and yaw to the camera while it is stationary at some position. **This is NOT to alter the world – the models' geometry and topology should not change**, only the camera's rolls around the specified axes. The camera may move after any of the three maneuvers, usually into the look direction, but can be different moves as well (e.g. repositioning in a circle or sideways moves). If you already have a flexible good camera that does things properly you can keep it.

9. (3) Allow 2D camera motion circling around the scene to observe the platform all around with a single key for each direction ("left" and "right"), e.g. arrow keys while in the rotation mode.

This can be either the main camera or a new separate camera initially positioned the same way as the main camera – therefore its radius is determined from its initial position. The camera is looking at the same initial look-at point throughout its travel around the circle. This means its up, right, and look-at vectors need constant updates as the camera position circles around the scene. You can use `gluLookAt()` or alteration of your `glPerspective()` or `glFrustum()`. **While strictly speaking all these operations work on the same model-view matrix, simulating this circling around with `glRotate*()` of the scene is not going to count.**

10. (4) Place a static camera at slightly in front of each of the 4 platform light sources "looking down" in the same direction as the light is cast. Allow viewing through each camera by the means of keys. Allow going back to the original camera view from Part 1.

These are new camera objects, distinct from any cameras before this point. Keep them separate. These cameras are always stationary.

These cameras' look-at point is identical to the street lights' light direction point, i.e. they both "look" at the same point on the floor.

Use 'R' to reset to the original viewing parameters of PA1.

11. (2) Place a camera in each robot (first person). Allow "seeing" through each such camera by using various keys on a keyboard as if seeing through their "eyes". A position, such as the light at the robots is sufficient, but the look direction is parallel to the floor.

These are distinct camera objects from the main and all previous cameras. They tag along with the characters at all times.

12. (3) Make sure all the keys are documented in detail in the help text invocable by the 'H|h' key. This is a **mandatory** requirement.

#### 4.2.1 Bonuses

1. (5) Allow tilting of the first-person view of the camera, i.e. both limited pitch and yaw motions so one can "look" slightly up or down or left or right without changing the current position of the camera.
2. (2) Allow a circling spotlight found at the same position as the circling camera and move along with it; with a turn on/off key.

### 4.3 Part 3 (PA3): Texture Mapping

The primary focus of this part is configurable texture mapping of your models. Marking with the bonus is  $(30 + 10)/30$ .

**IMPORTANT:** the textures (their underlying images) allowed that are either of your own production, or their license allows you to use them and redistribute them (cite the source), e.g. CC images from Wikipedia [5], but check the license of the images on Wikipedia or whatever is your source, including NeHe, OpenGL.org [6], NASA, and others.

Tasks:

1. (6) Model a skybox. A skybox is usually a gigantic cube large enough to surround your modeled world entirely, and it's usually texture-mapped on the insidewalls to simulate the far-out environment surrounding your main scene and often has a global light source where often the Sun is and the camera is restricted to the world boundaries. The platform is presumed to be in space, so the majority of the surroundings will be a stars. You will have six faces to texture map: (a) one has to have Earth-Moon system image, easily visible, (b) one has to have the Sun, also easily visible, (c) one has to have the NGC 281 nebula on it, (d) a sky with stars and your team number gently sprayed over it, (e) and (f) would be just plain sky with stars (distinct images), but you can use any other planetary bodies on them, nebulae, Milky Way, galaxies, etc. up to you. It's OK if the textures to not stitch very well on the edges.

Demonstrate the skybox in the wireframe. There are a lot of imagery available from space and ground telescopes and space agencies such as NASA, ISS, etc. to be used for the textures.

2. (19) Allow for the following textures by default for your models from Parts I and II. The requirements are purposefully “loose” to allow for creativity.
  - (a) (1) A golden pattern-laced texture covering the control unit.
  - (b) (1) The main base should have a texture of a spotted dark khaki military appearance.
  - (c) (1) The factories (just the building, not the component on top) should sport spotted light khaki.
  - (d) (1) The rubble piles (all types) should have textures of metallic debris.
  - (e) (5) Select five diverse textures of your choosing for the obstacles (construction blocks) on the platform.
  - (f) (1) Select a light metallic-looking texture pattern for the light posts.
  - (g) (1) Select rusty metallic-looking texture pattern for the floor tiles.
  - (h) (8) Pick a different metal (e.g. corten steel, aluminum, chromium, copper, etc.) for robot components than those of tiles, light posts, control units and debris. The components all may sport the same texture.
3. (1) Allow turning on and off the texture mapping (turning it off is roughly equivalent to your Part II’s lighting). A single key is to be there toggling the texture mapping on and off.
4. (5) Don’t forget your team number engravings when texturing.
5. (5) **Start** main project report document and the `README.txt` that describe your design, procedural modeling, software architecture, and the corresponding user manual. The `README` part would contain primarily the keys (the help output of your program), directory structure, and brief compilation instructions highlighting the multi-platform aspect of it if you have any. By **starting** it is meant to have the layout, initial structure with the user manual, some screenshots, title, all the team members (only names, email addresses, NO student IDs). (This document will have to be fully completed on/before the final project submission date.)  
At the demo time show the non-empty presence of the draft document of several pages with some of the mentioned material.

#### 4.3.1 Bonuses

1. (4) Model a skysphere instead of a skybox with the diameter equal to the skybox’s side length. If you develop both, allow switching between the two with a key. Demonstrate the sphere via wireframe.
2. (6) Distinct metallic textures per robot component (all chassis assume the same texture).

## 4.4 Final: Game Play, Level Maps, Advanced Features, Documentation

We build on top of the PA1–PA3 components. Bonus parts are optional, but will be counted towards the total for those who do them. Bonus parts, if completed, will have full weight only for

those who do them from scratch instead of porting any library or piece of code from elsewhere. Most points will be checked for their degree of presence between 0 and 1.

1. Adjust your project according to all the corrections in the previous parts if you did not already have them in. Expect to demo those points.
2. Allow a spotlight shining straight down from the control unit via a key. Pick suitable cut off angle and relevant properties.
3. Allow to change (alternate) robots' textures from the default (at least 2 "skins"). Provide a single key toggling through all the skin states. Document the key.
4. Attach a side view camera to the control unit that would follow it across the platform. The platform should scroll if not fitting into the screen (see crazy grid code example, [3]).
5. Allow "zooming in" into map objects by both, (a) either controlling and moving the camera towards them or (b) by keeping the camera in place and changing the `fovy` angle. Any available camera in theory can do the `fovy` changes while being stationary, but only the main camera, and perhaps the circling one (or both if they are one and the same) can do the motion with the position change. The cameras in characters do that naturally when the characters move.
6. Some minimal game play and scoring.
  - (a) One pre-stacked robot for your control (by default stationary) and one pre-stacked enemy robot-random with basic movement each time interval. They should face their direction of motion when move. The robots begin at their main bases.
  - (b) Motion of the control unit to be able to land on to your robot. When the control unit "mates" with the robot, the control keys begin controlling the robot instead.
  - (c) Minimal collision detection between the robots, bases, obstacles.
  - (d) Implement munition fire and collision detection for it. The robots have to have a number of hits threshold, when exceeded leads to robots destruction, conversion to a rubble pile, followed by a game over.
  - (e) At least two loadable maps. The opposing main bases should be at the opposite corners of the maps. The terrain in one of them should be mostly empty in the middle and the one one filled with obstacles (pits, rubble, blocks, factories) such that there is only a narrow passage between the bases. Keep the first one at 50x50 and the other one at 20x100.
7. Mouse-enabled camera motion of the main camera (enabled by a key).
8. Allow to toggle between full screen and windowed play.
9. Shadows. Basic (e.g. projection) shadows are a must for robots and other objects.
10. Complete the documentation as follows:

- (a) Title the documents in the form: “An “Open Robot Battle Near Earth” remake of “Nether Earth” by Team X: Brief Specification, Design, and User Manual, COMP371 Winter 2012”
  - (b) Do NOT include your student IDs, only your names and email addresses.
  - (c) The code must be thoroughly commented, especially the non-trivial parts.
  - (d) Fully complete the documentation you started in PA3.
  - (e) Describe your modeling methodology, animation techniques if any, texturing, OpenGL techniques and anything else relevant. Highlight the most difficult technical aspects and challenges encountered.
  - (f) Brief out your game’s software architecture, design decisions. Use any tools to briefly detail that information, such as UML or any suitable kinds of diagrams and the accompanying text.
  - (g) Document your modifications to any of the re-used source code components from the fellow open-source project(s), i.e. what parts are naturally yours and which came from the referenced project(s). Indicate also components that you did not re-use others’ code and wrote from scratch.
  - (h) Highlight and document all the extra things you performed above these minimum requirements.
  - (i) Provide some screenshots illustrating the key features of your game.
  - (j) **References to any resources used are of paramount importance.** Provide complete references section in the end.
11. **Bonus:** Use a static (non-interactive) ray tracer to render a scene snapshot.
  12. **Bonus:** Use curves (Super-ellipsoid), Bezier surface patches (dents in the models and components) to enhance your models of components, blocks, etc.
  13. **Bonus:** Use NURBS for the same.
  14. **Bonus:** Animate the robot destruction and conversion to a rubber pile either as morphing or falling pieces/particles.
  15. **Bonus:** More advanced game play (any or some or all of the following):
    - Radar map display (additional viewport with orthographic projection visualizing the current internal map).
    - Credit, time, robot, base, factory, part counts as resources and their production each time unit or spending.
    - Stats view with own and enemy robots, bases, factories, credits.
    - Multiple robots on all sides.
    - Lowering the control unit onto the home base to get a menu for the robot construction interface.

- Flags and occupancy of the factories and bases – standing in a base for  $X$  units of time “captures” the said base or factory.
  - Advanced AI for enemy robots.
  - Command interface (another orthogonal viewport) to make orders to the robots and script the orders, per original game.
16. **Bonus:** for those who did from scratch the **sliding platform** can be implemented to accommodate viewing scenarios when only a part of the platform is visible and if your robot or control unit reaches half-size of the platform into the invisible part’s direction, that part of the platform “slides” into the view. Those who re-used the posted code are ineligible for this bonus. The sliding in major part is achieved by affixing the camera at a constant distance to the control unit described earlier.
  17. **Bonus:** auto game play. Let the game play by itself with all the regular game play details, simple decision making, scoring etc. While it is progressing, allow viewing from all your cameras previously defined.
  18. **Bonus:** allow for adaptive level-of-detail based on the platform you are running to refine the game speed and the features to be playable at the reasonable speed. You can assign keys for that to either increase or decrease LOD. You can define LOD say as a set of booleans in an array that you cycle through with the said keys. At the end of the array all booleans are true meaning all features are enabled; at the beginning it is the most basic and fastest stuff. The indexes in the array can be set to mean to enable or disable a particular feature, e.g. you can encode gradual texturing in there of your world’s objects e.g. index  $X$  textures phases,  $Y$ , bipods,  $Z$ , for floor tiles,  $Z + 1$  for the wall textures, another index to control basic lighting vs. fully featured lighting, etc. Define constants describing what each array index corresponds to. In your draw or display function “teach” the code to look at these indices for the decision making to enable or disable a particular LOD feature.
  19. **Bonus:** allow team play with two robots, but different key sets to control the two (may need two viewports in the multiview mode).
  20. **Bonus:** Reflections. (Non-realtime and non-game play unless using the stencil buffer example). Just a frozen scene and the walls, the floor, the shiny pellets, and the street lights have the reflective properties. This bonus would count if you use environment mapping instead to simulate the reflections alongside the real-time game play.
  21. **Bonus:** the sound effects added to the game play would be a bonus. Like with images, either produce your own sound files, or make sure the ones you get off the Internet have an open-source or similar license that allows re-distribution. Cite such sources. The sounds would include shots, robot motion, destruction, etc.
  22. **Bonus:** portability of the source code between Linux, Windows, and MacOS X.
  23. **Bonus:** go as creative as you like beyond the minimum requirements. Feel free to enhance your models from external graphics packages you have access to, e.g. Blender [7]. The same applies to configurable visual enhancements of the textures, lights, the platform design,

alteration of the default game behavior, fancier animation, and so on. All such adjustments should be clearly indicated visually, in the code, and the documentation.

## References

- [1] Dick Grune, Brian Berliner, David D. 'Zoo' Zuhn, Jeff Polk, Larry Jones, Derek Robert Price, Mark D. Baushke, Brian Murphy, Conrad T. Pino, Fred Ulisses Maranh ao, Jim Hyslop, and Jim Meyering. Concurrent Versions System (CVS). [online], 1989–2011. <http://savannah.nongnu.org/projects/cvs/>.
- [2] Wikipedia. Concurrent Versions System — Wikipedia, the free encyclopedia. [Online; accessed 10-October-2011], 2011. [http://en.wikipedia.org/wiki/Concurrent\\_Versions\\_System](http://en.wikipedia.org/wiki/Concurrent_Versions_System).
- [3] Jeff Molofee and Contributors. Lesson 21: Lines, antialiasing, timing, ortho view and simple sounds. [online], Neon Helium (NeHe) Productions, 2000. <http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=21>; last viewed October 2011.
- [4] Peter Grogono. Concordia University Graphics Library (CUGL). [online], December 2005. <http://users.encs.concordia.ca/~grogono/Graphics/cugl.html>.
- [5] Jimmy Wales, Larry Sanger, and other authors from all over the world. Wikipedia: The free encyclopedia. [online], Wikimedia Foundation, Inc., 2001–2012. <http://wikipedia.org>.
- [6] OpenGL Architecture Review Board. OpenGL. [online], 1998–2011. <http://www.opengl.org>.
- [7] Blender Foundation. Blender. [online], 2008–2011. <http://www.blender.org>.