TSBK07 Computer Graphics
Project Report for
NQ Sokoban

Erik S. Vasconcelos Jansson
erija578@student.liu.se
Linköping University, Sweden

May 25, 2016

1 Introduction

This document describes a preliminary project specification for the logic puzzle game *NQ Sokoban*, which is a game based on the classical *Sokoban* game described in Section 1.1 and on the aesthetics of the game *EDGE* described shortly in Section 1.2. However, there are some gameplay twists. Therefore, *Not Quite Sokoban* is described in Section 1.3.

1.1 Sokoban

The very popular and classic logic puzzle game. *Sokoban* is a game where the player is tasked to *push* stones/crates/boxes to a *goal area* with as *few moves* as possible. Originally released and invented in 1982 [Lin98] by *Hiroyuki Imabayashi* of *Thinking Rabbit*, a game studio in Takarazuka, Japan. The original game became available in Europe around 1984 to the *Apple IIe*, published by *Spectrum Holobyte Inc.* Interestingly, the name *Sokoban* literally means *warehouse man* in Japanese.

1.2 EDGE (iOS)

A puzzle game developed and published by *Mobigame* in 2008 [Wik15] for the *Apple iPhone*. *EDGE* is a game where one guides a cube around different levels collecting as many prisms as possible before reaching the end of a level. By swiping on the screen the player can move the cube and *also climb surfaces* by balancing on them. The aesthetics are very minimal, only featuring a *voxel world*. See [Wik15] for more information (and gameplay).

1.3 NQ Sokoban

Rather than being a straight up 3D Sokoban clone (there are already a lot of those), the game project *Not Quite Sokoban* tries to innovate somewhat by adding a few extra gameplay twists/mechanics.

Besides the original rules of Sokoban described in Section 1.1, additional *layers* (an extra dimension that can be traversed) are added to the world/level. These create interesting implications, especially if disallowing *blocks* to be pushed up a layer but still allowing them to fall down several of these. An additional mechanic that can be envisioned is to disallow traversal through a path that has an abyss (meaning that there are no blocks to fall down to) *but* if a *goal area* is there, the player is allowed to build a *"bridge"* by pushing a movable block there (gaining access to a new area). See Figure 2 for e.g.

The above changes to the gameplay mechanics do not affect normal Sokoban levels, so *NQ Sokoban* is essentially "backwards compatible" (see Figure 1) with the original Sokoban levels given that these only use compatible mechanics (2 layers). Additional mechanics could be added later, but additional complexity would make the game inelegant.

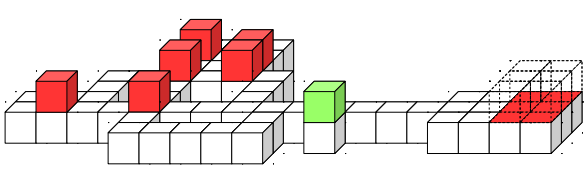


Figure 1: Concept/Draft of Aesthetics

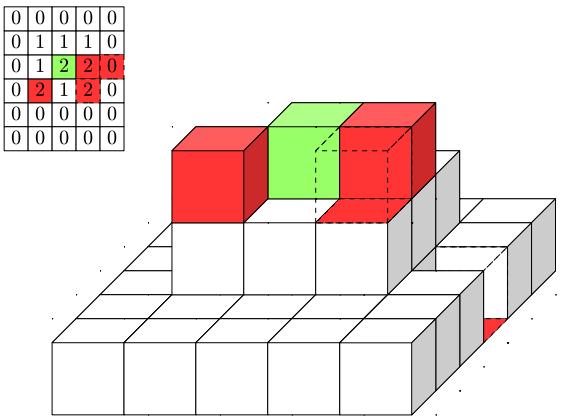


Figure 2: Some 3D Gameplay Twists

2 Requirements

2.1 Obligatory

- Voxel-like world representation.
- Colors vary depending on object type.
- Rules of NQ Sokoban & original Sokoban.
- Blinn-Phong [Rag08, p. 66] reflection model.
- Flat shading [Rag08, p. 69] for these.
- Collision detection (part of the rules).
- Loading levels via JSON or similar.
- Animation for player movement.

2.2 Probable

- Modifiable "color themes" with level.
- Packs of levels, also in JSON or similar.
- Overlay with name of level and current moves.
- High score list for level (or level pack).
- Different camera modes (e.g. top view).
- Customizable textures for objects (not colors).

2.3 Luxury

- Screen-space ambient occlusion.
- Sound effects for pushing and winning.
- Neat transition between levels (with score).
- Bump mapping for customizable textures too.

3 Technologies

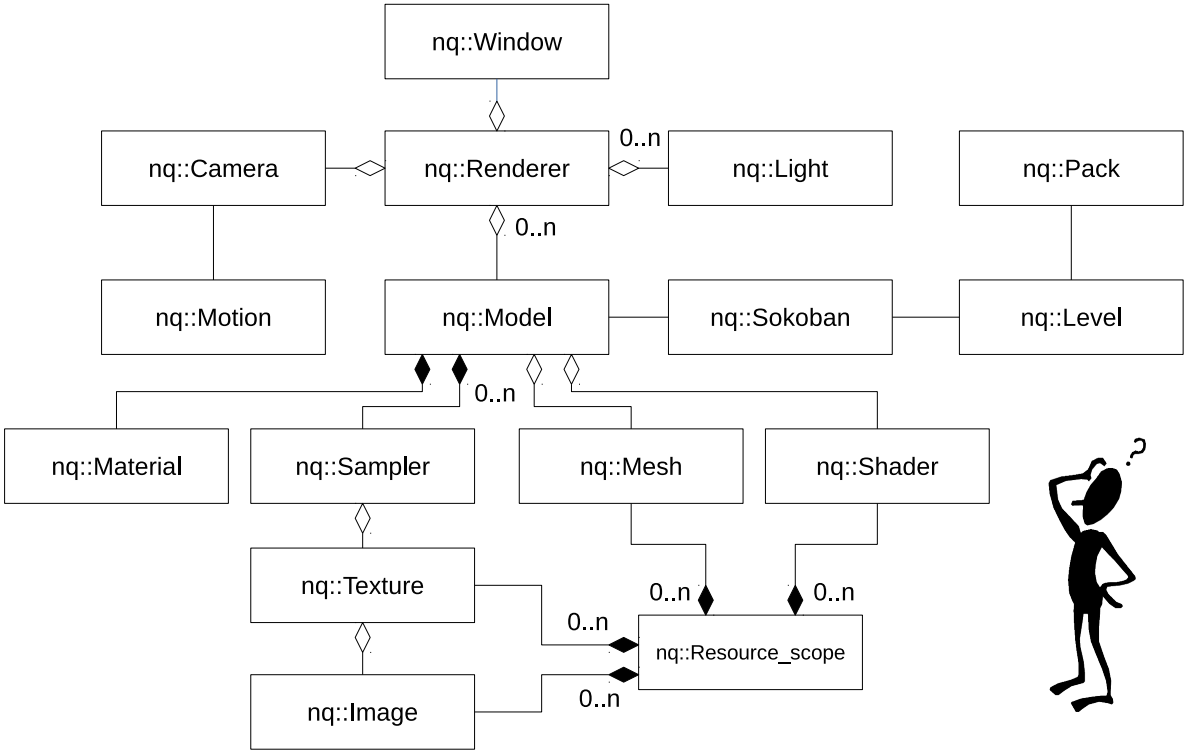
- **C/C++:** since it performs quite well.
- **OpenGL:** for actual computer graphics.
- **GLFW**: window/context creation and input.
- **GLEW**: extension wrangler for latest GL.
- **GLM**: nice interaction between GLSL.
- **JSONCpp**: for parsing JSON text files.
- **libPNG**: for loading textures (if any).
- **libOGG, libVorbis & OpenAL:** audio.
- **FreeType**: for loading font glyphs.

4 Implementation

Before actually starting to build the game itself, it was reasoned that a *game framework* be done first. Doing this would enable easier development and modification of the game later, since two different parts would be built, the *framework* and the *game*, decoupling them from each other, leading to a general framework that can be used for future project or extensions, and a game that wouldn't be tightly coupled with the nasty low-level moving parts of it.

Additionally, it was decided that the game would also be subdivided into two parts, leading to higher cohesion and less coupling, with the primary *game visualization* and *game logic* subsystems separated. Allowing "hot-swapping" of modules with a lower amount of effort. Therefore, *game logic* is only used to update the game rules according to states received, while *game visualization* issues queries of model locations to draw based on it, and only that.

Rough overview of the combined architecture of the *game framework*, *game visualization* and *game logic* follows in the coming page within a diagram. All of this can be found in the attached source code.



4.1 Game Framework

Basically contains a series of useful *wrappers* for various different *foreign libraries* and then builds on these to build abstractions suitable for a complete *game framework*. `Window` abstract a lot of the details and management in creating a platform independent window and OpenGL context with *GLFW*. `Renderer` then uses it to supply OpenGL settings, where it is also later used for *drawing some models*.

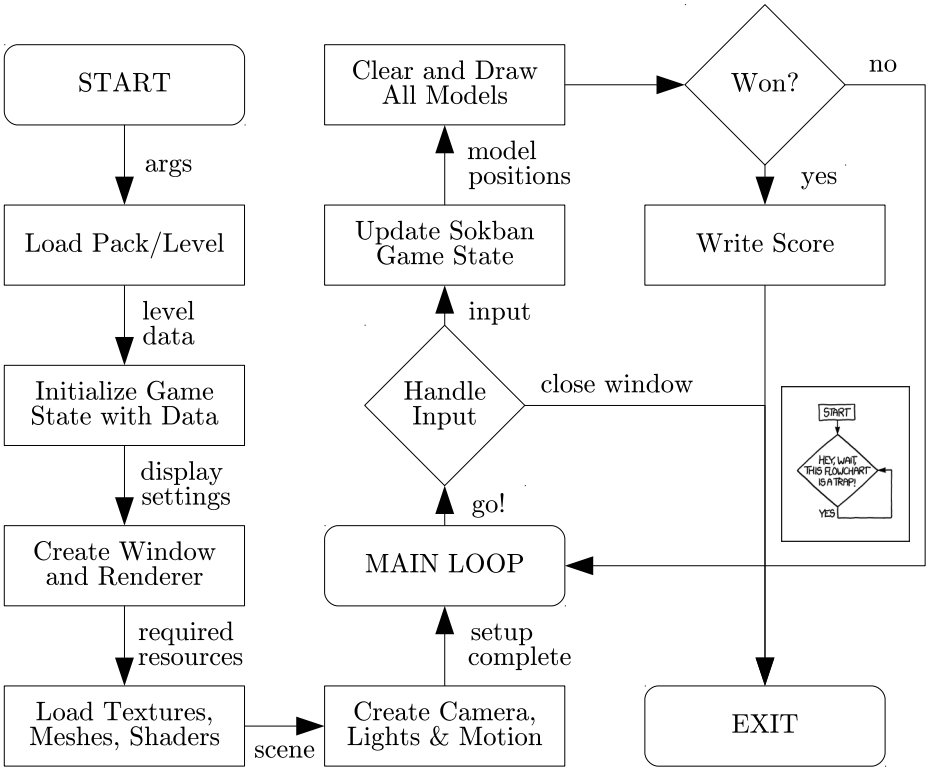
Before doing so though, there are three essential components which are necessary to describe a suitable *game scene*: `Camera`, `Model(s)`, `Light(s)`. Using these, the renderer can only draw one model, where several subsequent draw calls are acceptable before issuing a `display` command to the window for *swapping buffers*. Animations can be specified with the `Motion` package, which contain several *Robert Penner's easing equations* [Pen02], which are used to specify predetermined motion patterns.

Perhaps the most central module is the `Model`, which is just *glue* for uniquely specifying several *visual attributes* of the model. `Material`, `Texture Sample(s)`, `Mesh` & `Shader(s)` are used there.

4.2 Game Mechanics

After the *framework* is up and running, we can both *draw* and read *input*, which is all that is necessary. Essence of the game *rules* and *logic* are contained within the `Sokoban` class, which is given *level data* as input as a precondition for before starting. Data is essentially one layered cake of two-dimensional arrays, each specifying one height level. By using this data, together with the *action* the player wants to perform, the game will `step` the state of the world forward, while applying all of the game rules. For example, collision is just a forward array check. Additionally, the player should be able to *undo* or *reset* his actions, which can be done since the game keeps track of previous state by the means of a stack, which can be unwound up to a previous state.

However, how does one load new levels before the same old one gets boring? By using the `Level` module, *files* are read into memory and parsed through a *JSON parser*, which gives all of the *meta information* needed. The level data itself is contained within image files, which are loaded and merged using *libpng*, and then passed to `Sokoban`.



4.3 Game Flow

Now follows a brief description on how all of these modules actually develop and interact over time. See the flowchart above, it's mostly divided in three phases: *setup*, *main loop* and *exit*. *Setup* consists of initializing windows, renderers and game states, while loading in all required resources for the scene. Loading in *level data* and converting it to a *mesh* required a custom class, a `Mesh::Builder`, which merges together several *voxels* into one *single mesh*.

After the scene has been successfully setup, the game starts listening for user input, and updates the internal `Sokoban` game state accordingly or, if *exiting*, closes closes down/frees all open resources. By using these states, the game can determine where to position the more dynamic parts of Sokoban, which are the *player, crates/moveables* in the correct world position, by transformations. With this, the *model* can be draw to the buffer, which applies the *reflection calculations* within the *vertex shader*, which implies fast *Gouraud shading*.

Finally, once the game state has determined the player has won, best scores are stored down to disk.

4.4 Problems

Since this is a project, several issues were encountered and solved. Primary ones are listed therefore:

- **Draw calls:** having huge or even medium sized levels caused performance problems since the overhead of doing single draw calls for *all* voxels taxed the bus greatly, which was solved by merging voxels ussed for the level into one single mesh, only requiring on single draw call.
- **Phong shading:** since the target hardware was old (Intel GMA 4500), pumping through the *reflection model operations* on each fragment became unfeasible, and would slow down the game considerably when the camera was close to the geometry, even for small levels, so *Gouraud shading* was employed by moving these calculations to the *vertex shader* instead.
- **Resource handling:** even with C++'s RAII, certain resources needed to be managed with a `Resoure_scope` wrapper class to model the dependencies of e.g. *meshes* to the *data buffers*.

5 Conclusions

A fully working games has been implemented, which is both mechanically unique and expandable. While several aspects could have been improved, such as dynamic shadows or ambient occlusion, the game accomplishes an non-intrusive aesthetic feel. Below follows a list of accomplished requirements:

- **Voxel world:** accomplished, done by loading level data through the custom file format and then merging together a static mesh geometry.
- **Varying colors:** accomplished, by using the uniform variables, these could be changed on the fly to reflect the game state & object types.
- **Game rules:** accomplished, simply uses the loaded level information to check that given input actions are consistent with the game rules.
- **Flat shading:** accomplished, uses the Phong reflectance model in vertex shader, but has flat shading anyway since the voxel normals remain constant even after GLSL interpolation.
- **Collision detection:** accomplished, simple, just check the loaded array and apply game rules to see if player/moveable is colliding with anything of relevance, if so, don't apply action.
- **Loading levels:** accomplished, by using an external library for parsing JSON, the custom file format was fairly straight forward, where level packs can also be specified along with score data and relevant level meta information.
- **Animations:** sort of accomplished, camera has smooth transitions between states, however, game objects are still moving in voxel steps, which should also have been achieved :(
- **Themes & Packs:** player can specify what color the different game objects should be for his/her own level, which is done via the level file format. Also, level packs, can be specified.

Future work should be focused on improving the aesthetic depth of the game with better shadows, smooth object interpolation between moves, use by having a graphical menu and interface with texts, loading Sokoban files would be nice out-of-the-box. Finally, for the final polishes, non-intrusive sound effects and an integrated level editor would be nice.

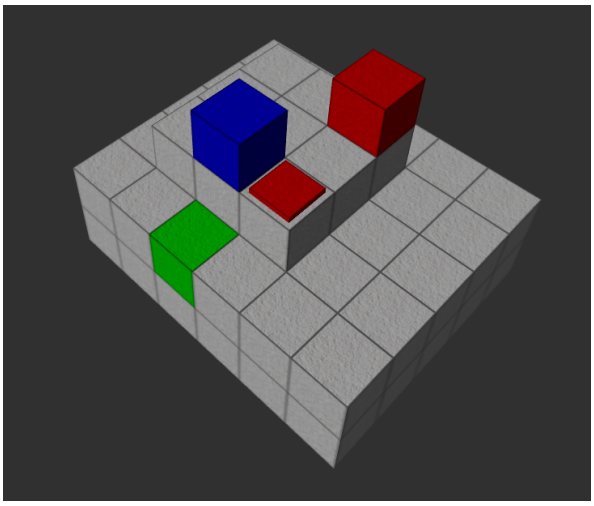


Figure 3: Screenshot of the Current Build

References

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-time rendering*. CRC Press, 2008.

[DP11] Fletcher Dunn and Ian Parberry. *3D math primer for graphics and game development*. CRC Press, 2011.

[Eri04] Christer Ericson. *Real-time collision detection*. CRC Press, 2004.

[Lin98] Scott Lindhurst. *Sokoban: History, levels and other implementations*. Online documentation, Princeton University, 1998.

[Pen02] Robert Penner. *Robert Penner's Programming Macromedia Flash MX*. McGraw-Hill, Inc., 2002.

[Rag08] Ingemar Ragnemalm. *Polygons feel no pain*, volume 1. Bokakademin, 2008.

[Sch14] Jesse Schell. *The Art of Game Design: A book of lenses*. CRC Press, 2014.

[Wik15] Wikipedia. *Edge (video game), Gameplay*, 2015. [Online; accessed 23-January-2016].