Programming in Haskell — Fall 2019
Project descriptions: Stage I

Nickolay Kudasov

Contents

1 About projects **3**
- 1.1 Stage I . . . 3
- 1.2 Stage II . . . 4
- 1.3 Submissions . . . 4
- 1.4 Q&A and discussions . . . 4
- 1.5 Coding style . . . 5
- 1.6 Prohibited project ideas . . . 6

2 Interpreters & Compilers **7**
- 2.1 λ-calculus . . . 7
- 2.2 A Lisp dialect . . . 9

3 Embedded domain-specific languages **11**
- 3.1 Mendelian inheritance . . . 11
- 3.2 Music composition . . . 13

4 Puzzles **15**
- 4.1 Pipes . . . 15
- 4.2 Transport control puzzles . . . 18

5 Games **20**
- 5.1 Arkanoid . . . 20
- 5.2 Board games . . . 20
- 5.3 Endless runners . . . 22

5.4 Tower defense 23

6 Simulations **25**
6.1 Fluid simulation 25

7 Other ideas **28**

1 About projects

1.1 Stage I

The first stage of the project is focused on getting some minimal thing to work. This stages is done in teams of up to 3 people and the code is the same. It does not matter who actually wrote the code as long as every member of a team has contributed to it and is fluid in how the code works.

Different teams cannot share the same project!

This means that you have to approve your choice with the principal instructor before you can work on the project. Sometimes variations of the same project can be approved for several teams if those variations differ deeply enough.

Grading will be performed by giving each teammate a little personalised task that should be accomplished in 15–30 minutes in class and will typically require to either fix something, add a small feature or modify the code slightly.

1.1.1 Recommendations

Start your project by deciding what features you want in your project *as a team*. Writing down informal specification or possible features for the future (brainstorm ideas) might be useful.

Next determine the core data types that you'll be needing to model the domain accurately. Try to come up with datatypes that closely model your domain and try make illegal states unrepresentable.

Next think of the core functions that implement the logic for your domain. **Keep your logic pure!** Documenting functions with examples and detailed explanations might be helpful.

Now you can put things together into an application. In Stage I you don't need much `IO`, so stick to simple stuff.

For graphics you can use CodeWorld, but it will not be enough for Stage II. So instead you might want to try a similar library called `gloss`. See function `play` from `Graphics.Gloss.Interface.Pure.Game` which almost the same as `interactionOf` from `CodeWorld`.

1.2 Stage II

Stage II will build on top of Stage I, but will add more depth to the project. In Stage II every teammate will have an individual assignment, although cooperation is highly encouraged to bring new features together and make the project significatly more interesting and complete.

You don't have to pick the assignment right now, but you can think of it when considering what you want to build in the end.

Two teammates cannot have the same individual assignment!

This requirement is similar to that for Stage I.

1.3 Submissions

You are encouraged to use a Git repository, for instance one hosted on GitHub or Bitbucket with access for all teammates (or in public domain). A submission would then be a commit (fixed state) in that repository. However, you can also submit your project as an archive with all the source code and necessary files.

You are allowed to share your project source code with your classmates and in public, however, you will have to provide some evidence that you are the author.

Grading of submissions will be done in-class, where every teammate will be asked to modify the program in some way to prove their familiarity with the codebase.

1.4 Q&A and discussions

If you have any questions regarding project, feel free to ask:

- in the course Telegram group chat,
- directly via Telegram: @fizruk31337,
- directly via e-mail: n.kudasov@innopolis.ru,
- before, during and after classes.

1.5 Coding style

Write your programs creatively using these constraints as a guide to the beautiful:

- Use `camelCase` to name functions, variables and types.
- Use descriptive names. Let them be as long as needed, but no longer that that. Good: `renderRemaining`. Bad: `rndr`. Ugly: `renderObjectsWeHaven'tRenderedYet`.
- Do not use tabs: Haskell is sensitive to indentation, so tabs can ruin your experience. Configure your editors so that they convert TABs into spaces automatically.
- Try to keep lines at 80 symbols or less. It is normally easier to read code if you don't have to scroll horizontally.
- Specify a type for every top-level function. This improves documentation, helps your thinking and improves compiler error messages. Usually there is no need to specify types for locally defined functions (e.g. with `let` or `where`).
- Provide a comment for every top-level function, explaining what that function does. Sometimes examples can help a lot with that explanation.
- Use `-Wall` or `{-# OPTIONS_GHC -Wall #-}`. This flag turns on a lot of useful warnings.
- Define all functions total, i.e. defined on all inputs. If a function crashes or loops on some input — it is not total.

1.6 Prohibited project ideas

These projects are not allowed for implementation either because they are too simple or because there exist multiple implementations on the Internet:

- checkers;
- chess;
- reversi;
- tic-tac-toe;
- asteroids;
- chainword (linear crossword);
- sudoku;
- nonogram (Japanese crossword);
- Conway's game of life.

2 Interpreters & Compilers

2.1 λ-calculus

λ-calculus (lambda-calculus) serves as the foundation for many functional programming languages: Lisp family (Common Lisp, Scheme, Clojure, etc.) and ML family (Standard ML, Haskell, OCaml, F#, etc.)

λ-calculus consists of a language of λ-terms and a set of transformation rules.

Basic syntactic rules for λ-calculus are these:

- a variable x is a λ-term;
- if e is a λ-term and x is a variable, then $(\lambda x.\ e)$ is also a λ-term (**λ-abstraction**);
- if e_1 and e_2 are λ-terms, then $(e_1\ e_2)$ is also a λ-term (**application**).

For convenience the following simplified notation can be used:

outermost parenthesis can be omitted

$$(\lambda x.\ e_1)\ e_2$$

application is left-associative

$$e_1\ e_2\ e_3 \equiv ((e_1\ e_2)\ e_3)$$

λ-abstration body extents to the right as far as possible

$$\lambda x.\ e_1\ e_2 \equiv \lambda x.\ (e_1\ e_2)$$

nested λ-abstractions can be written as one

$$\lambda x.\ \lambda y.\ \lambda z.\ e \equiv \lambda x\ y\ z.\ e$$

Operator λ *binds* variable x in $\lambda x.\ e$. Variables that have are introduced in a λ-abstraction are called *bound*. All other variables are called *free*. For instance, variable y is free in $\lambda x.\ y\ x$. Variable is bound by the *closest* λ-abstraction. For instance, in $\lambda x.\ y\ \lambda x.\ x$ the only occurence of x is bound to the second λ.

The evaluation of λ-terms is defined by the following reduction rules:

α-conversion renaming of the bound variables

Example: $\lambda xy.zxy \equiv \lambda ab.zab$

β-reduction application of a function to its arguments

Example: $(\lambda s\ z.\ s\ (s\ z))(\lambda n.\ n) \equiv \lambda z.(\lambda n.n)((\lambda n.n)z)$

η-conversion expresses the principle *two functions are identical iff they are identical on all inputs*

Example: $f \equiv \lambda x.\ f\ x$

Function application is implemented by substituting an expression in every place where a bound variable occurs in the body of a *lambda*-abstraction:

$$
\begin{aligned}
x[x \mapsto e] &\equiv e \\
y[x \mapsto e] &\equiv y, \text{if } y \neq x \\
(e_1\ e_2)[x \mapsto e] &\equiv (e_1[x \mapsto e])\ (e_2[x \mapsto e]) \\
(\lambda x.\ e_1)[x \mapsto e] &\equiv \lambda x.\ e_1 \\
(\lambda y.\ e_1)[x \mapsto e] &\equiv \lambda y.\ (e_1[x \mapsto e]), \text{if } y \neq x \text{ and } y \text{ is not a free variable in } e
\end{aligned}
$$

The language of λ-terms can be extended with

- base data types (e.g. numbers, boolean values, strings);
- builtin container types (lists, tuples, array, mutable references, etc.);
- builtin operations and functions (e.g. $+$, $-$, sin, cos, *and*, *or*, *concat*)
- special syntactic constructions (e.g., $if \ldots then \ldots else \ldots$ or $let \ldots = \ldots in \ldots$);
- static or dynamic type system;
- user-defined data structures;
- etc.

2.1.1 Minimal requirements

Minimal implementation should have:

- a data structure for the abstract syntax tree of λ-terms;
- a parser, independent from interpreter;
- an interpreter (with any reduction strategy);
- a simple read-eval-print-loop (REPL);
- at least two basic data types with a few builtin operations;
- a sample program.

2.1.2 Stage II preview

In stage II you can extend λ-calculus to build a simple functional program on top. Individual assignments may include:

- fault-tolerant parser and human-friendly error messages;
- λ-calculus extensions:
  - type system;
  - evaluation strategies;
- code generation (compilation);
- visualisation tools.

2.2 A Lisp dialect

In this project you need to implement a Lisp programming language dialect.

Programs in Lisp are composed of S-expressions and in this dialect it can be:

- an atom (`IAMATOM`, `numberp`, `setf`),
- a numeric literal (`10`, `34.2`),
- a string literal (`"hello, world!"`),
- an empty list (`nil`, `()`),
- a dotted pair (`(1 . 2)`).

As usual for Lisps, non-empty list is represented by a dotted pair, where the second component is the tail of that list: `(1 "asd" atom)` is equivallent to `(1 . ("asd" . (atom . nil)))`.

All S-expressions can be evaluated by the following rules:

- string and numeric literals, empty list and atoms evaluate to themselves;
- a list `(f ...)` evaluates by applying a function `f` (or expanding a macro `f`) to the arguments (which are the rest of the list);
- function arguments must be evaluated before application;
- macros and special forms do not evaluate their arguments.

A program is a sequence of S-expressions.

This dialect can be extended with

- builtin special forms:
  - `quote` (`'`);
  - `if`;

  - `let`;
  - `macro`;
  - `setf`;
- used-defined functions: `lambda`, `defun`;
- lexical/dynamic binding;
- input-output operations;
- polyvariadic functions (functions that can work with variable number of arguments);
- etc.

2.2.1 Minimal requirements

Minimal implementation should have:

- a data structure for the abstract syntax tree of λ-terms;
- a parser, independent from interpreter;
- an interpreter (with any reduction strategy);
- a simple read-eval-print-loop (REPL);
- at least the following builtin predicates and functions:
  - `atomp`, `numberp`, `listp`, `=`;
  - `car`, `cdr`, `cons`;
- mechanism for defining user macros or functions;
- sample program (e.g. tic-tac-toe).

2.2.2 Stage II preview

In stage II you can extend λ-calculus to build a simple functional program on top. Individual assignments may include:

- fault-tolerant parser and human-friendly error messages;
- Lisp dialect extensions:
  - standard library;
  - macros and functions;
  - map-like structures and lazy sequences (a la Clojure);
  - type system (a la Hacket or Typed Clojure);
- code generation (compilation);
- visualisation tools.

3 Embedded domain-specific languages

3.1 Mendelian inheritance

In this project you're required to implement a Haskell library presenting an ergonomic way to work with genetics problems using Mendelian inheritance laws.

Mendelian inheritance allows one to predict traits for future generations based on the following laws:

Law of Segregation of genes (the "First Law")

- Every individual organism contains two *alleles* for each trait, and that these alleles segregate (separate) during meiosis (cell division) such that each gamete contains only one of the alleles. Each offspring thus receives a pair of alleles for a trait by inheriting one allele from each parent.

Law of Independent Assortment (the "Second Law")

- Alleles for separate traits are passed independently of one another.

Law of Dominance (the "Third Law")

- Recessive alleles will always be masked by dominant alleles. When both types of alleles are present, only a trait encoded by the dominant one will show. So if one of the parents has two dominant alleles, then first generation of offsprings is guaranteed to have a dominant trait.

In the image above the alleles are represented using capital and small letters. For instance capital `S` (short tail) represents an allele encoding the dominant short tail trait and small `s` represents a recessive allele for a long tail. And capital `B` (brown) represents a dominant allele encoding brown fur color while small `b` represents a recessive allele for white fur color.

Genotype is a set of alleles for all traits (e.g., **AAbbCc**).

Phenotype is a set of actual traits that an organism has (e.g., **AbC**).

Common problems that can be solved with Mendelian inheritence are as follows:

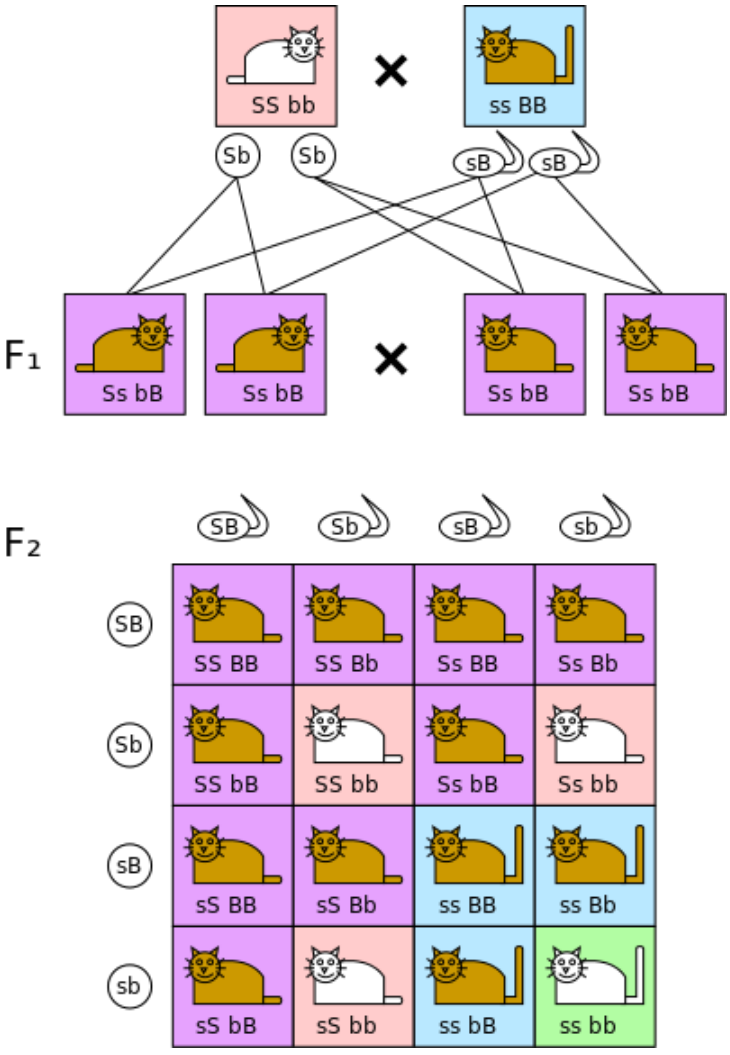


Figure 1: Dihybrid cross.

- determine phenotypic ratios for an n-th generation given parents' genotype(often also as ratios);
- guess parents genotype given parents and children phenotypes.

The library should provide ergonomic means to solve such problems using user-provided traits.

3.1.1 Minimal requirements

Minimal implementation should have:

- data structures for genotype, phenotype and corresponding ratios to describe distributions of genes and traits in generations;
- a way for a user to define genotypes;
- a way for a user to use their own traits;
- a function to compute the first generation of offsprings;
- a sample problem and its solution using implemented library.

3.1.2 Stage II preview

In stage II you can extend the library with more useful features. Individual assignments may include:

- non-Mendelian inheritance (multiple alleles, co-dominance, polygenic traits, etc.);
- different cross strategies (e.g. include parents in both 1st and 2nd generations);
- guess parent genotypes by phenotypic or genotypic ratios for (any) generation;
- visualisation with branch diagrams or Punnett squares;
- a interactive graphical user interface.

3.2 Music composition

In this project you need to implement a Haskell library for composing or manipulating music.

Core entities for this library are

- note (C, D, E, F, G, A, B with alterations: ♯, ♭ and ♮);
- rest (e.g. 𝄽, 𝄾, etc.);

- interval (diatonic and chromatic);
- duration (𝅝, 𝅗𝅥, ♩, ♫, etc.);
- scale (a sequence of intervals, e.g. major scale, minor scale, blues scale, etc.);
- harmony (chords, background for a melody).

The library has to offer some ergonomic ways to create melodies and parts, to combine small pieces into more complicated parts and compositions.

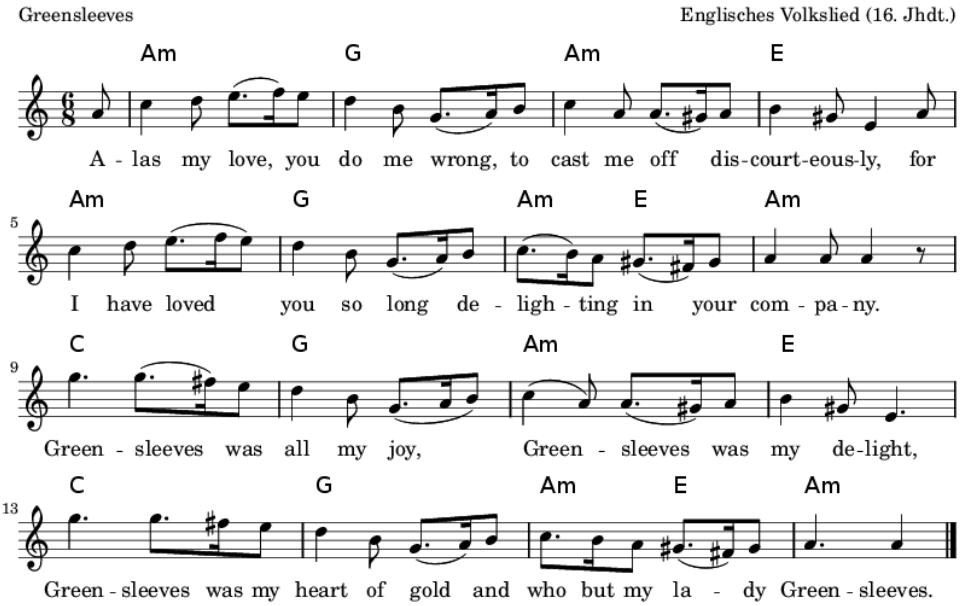


Figure 2: «Green sleeves».

3.2.1 Minimal requirements

Minimal implementation should have

- means to encode a simple one-voice melody (with rests);
- transposition (shifting all notes by a given interval);
- sequential concatenation of melodies;
- parallel concatenation of melodies (making multi-voice parts);

- some way of visualising melodies, more comprehensible than a `Show` instance;
- a sample multi-voice melody built with implemented library.

3.2.2 Stage II preview

In Stage II you could extend the library to support more tools, analyse or visualise music, output into different formats. Individual assignments may include:

- jazz harmony, generated improvisation;
- harmony analysis: guess harmony from melody;
- support for song lyrics;
- support for output in LilyPond, MusicXML or MIDI;
- an interactive graphical user interface for music notation;
- extended music notation (triplets, measure, keys, bars, guitar tabs notation, percussions, etc.);
- performance visualisation with animated piano keyboard.

4 Puzzles

4.1 Pipes

Pipes is a puzzle where you build a path for a water stream by replacing tiles with pipes. Usually a game consists of a bounded tile map filled with random tiles and with source and sink tiles. There may be multiple source/sink tiles in some variations. The goal is to complete a path from source to sink before the water reaches an open end of some pipe and leaks.

There are a few tiles to choose from and when some tile is used it is replaced with a random new one. There may be an infinite or a finite supply of tiles.

4.1.1 Minimal requirements

Minimal implementation should have:

- minimal number of tile types that is enough to complete a puzzle;
- prefilled or randomly generated map of size at least 10×10;
- an infinite supply of tiles (possibly predefined);

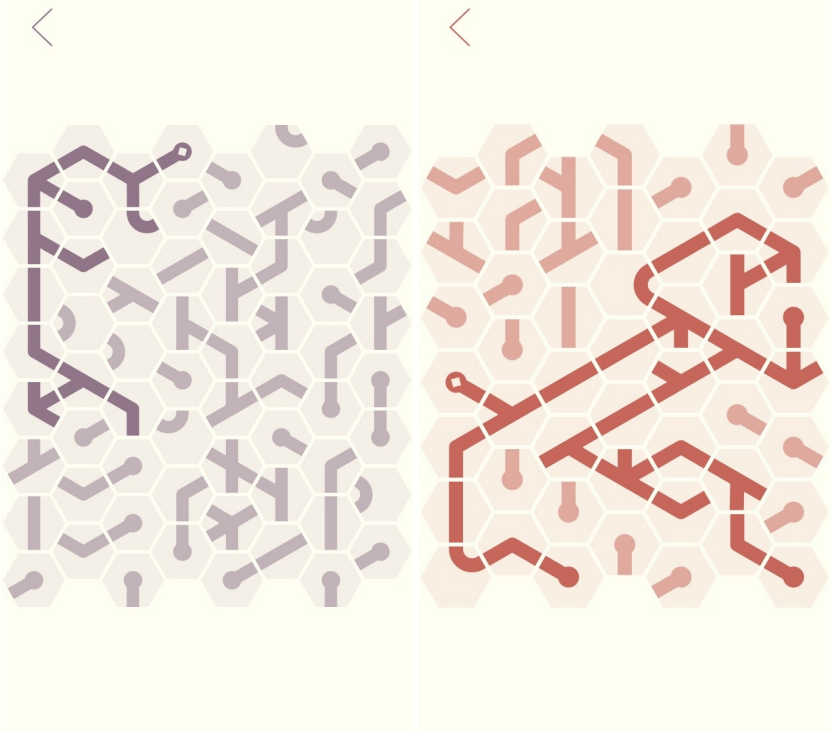


Figure 3: Noodles!

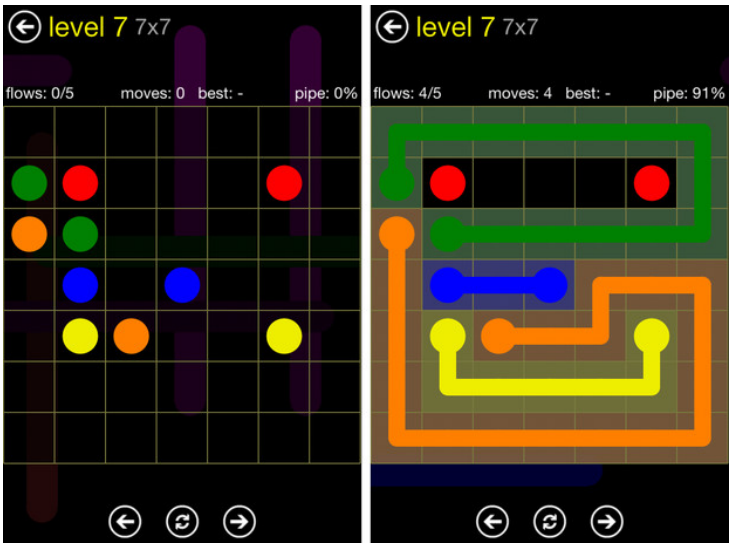


Figure 4: Flow

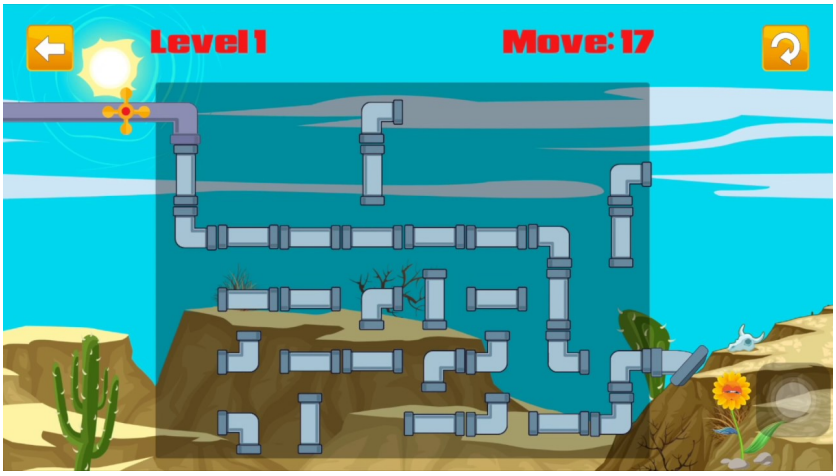


Figure 5: Plumber Pipe

- water flow animation;
- water leakage detection (game over);
- victory detection.

4.1.2 Stage II preview

In Stage II you can make the game more interesting with extra features, multiple levels, advanced user interface, multiplayer and more. Individual assignments may include:

- extra features:
  - bonuses, pressure-sensitive pipes, vortices, etc.;
  - multiple (possibly matching) sources and sinks;
- multiplayer;
- advanced GUI:
  - menu;
  - virtual or augmented reality interface;
- 3D tiling;
- client-server architecture.

4.2 Transport control puzzles

Transport control puzzle is an interactive puzzle where you build and/or control a transportation system to satisfy the evergrowing demand.

For instance, in Minimetro-like game you need to continuously build an underground transportation system by building rail lines connecting stations, adding new trains or improving other things.

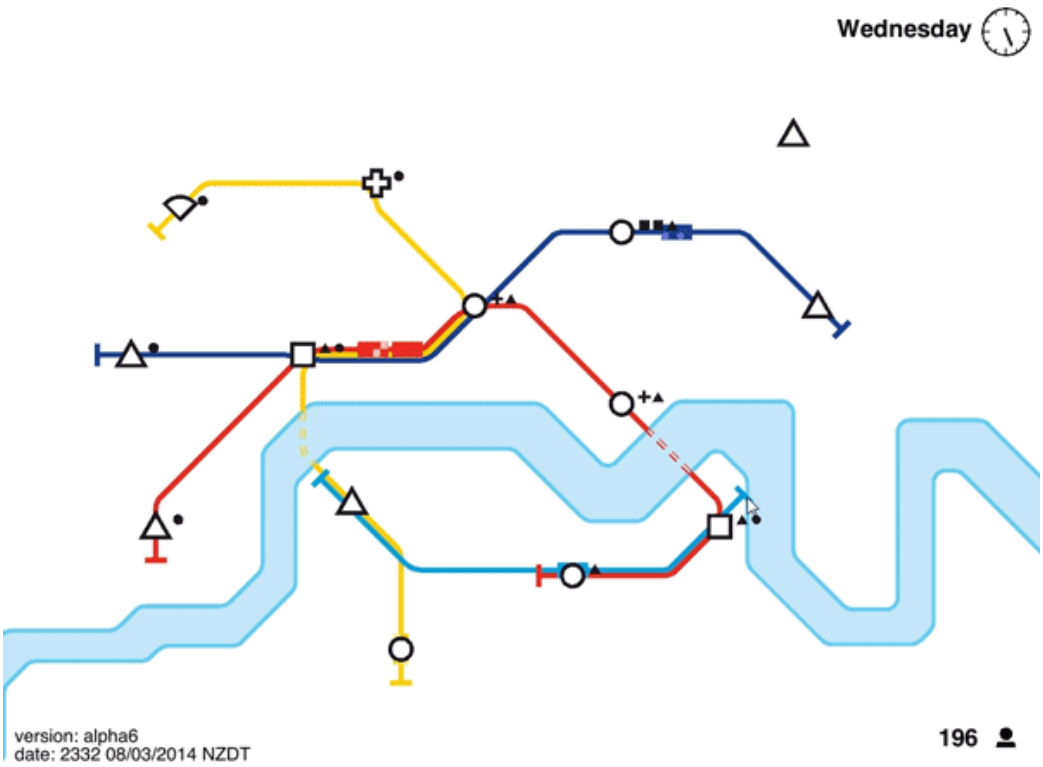


Figure 6: Minimetro

In a "Flight Control"-like game the player assumes the role of a traffic controller, making sure planes don't collide and all planes and helicopters land.

4.2.1 Minimal requirements

Minimal implementation should have:

Figure 7: Flight Control

- some trade-off mechanics (a choice for a player to make, e.g. build a line or add a train);
- very basic implementation (one type of passengers/planes, fixed map);
- infinite gameplay (passengers/planes arrive constantly until game is over);
- transport system collapse detection (game over, e.g. overcrowded stations or plane crashes).

4.2.2 Stage II preview

In Stage II you can extend transport system to closely mimic a realistic one, add advanced user interface, multiplayer and more. Individual assignments may include:

- different game modes (local, network, AI/Human, etc.);
- realistic distributions for traffic;
- advanced GUI:
  - menu;
  - virtual or augmented reality interface;
- multiplayer;

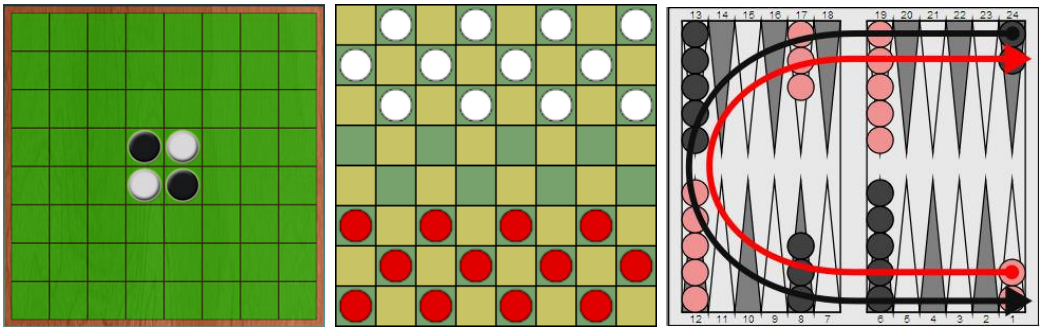
- artificial intelligence solving traffic problem;
- client-server architecture.

5 Games

5.1 Arkanoid

5.2 Board games

In this project you are required to implement a board game (such as reversi, chess, backgammon, go, etc.).



Some more original board games:

- «Kamisado» (similar in complexity to checkers);
- «Hive» (similar in complexity to chess).

5.2.1 Minimal requirements

Minimal implementation should have

- basic graphical interface with Human vs Human mode;
- only valid moves allowed;
- a mechanism to determine the end of the game (victory or tie);
- a mechanism to display game outcome (e.g. who won).

5.2.2 Stage II preview

In Stage II the game can be extended with more rules or variations, an AI, an advanced interface or client-server architecture. Individual assignments may include:

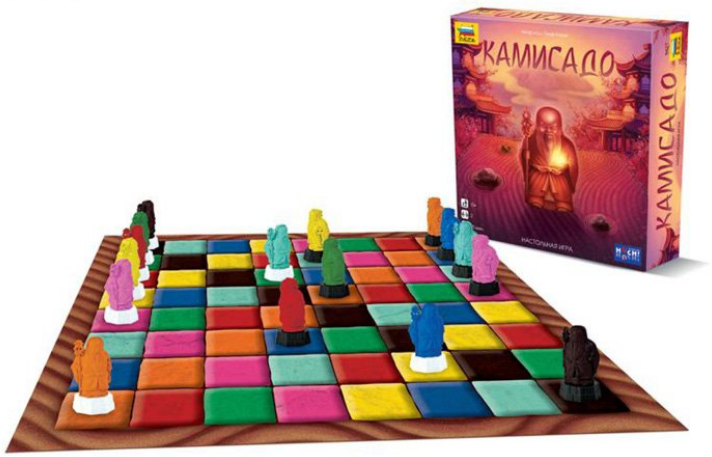


Figure 8: Kamisado.

Figure 9: Hive.

- different game modes (local, network, AI/Human, etc.);
- advanced GUI:
  - visualisation for possible moves;
  - menu;
  - virtual or augmented reality interface;
- a configurable artificial intelligence for an opponent;
- client-server architecture.

5.3 Endless runners

An endless runner is a computer game with an infinite, procedurally-generated world (usually a side-scrolling platformer) where the primary goal is to last as long as possible.

In this project you have to implement some variation of an endless runner.

Figure 10: Worm Run 2.

Figure 11: Tiny Wings.

5.3.1 Minimal requirements

Minimal implementation should have

- a simple procedurally-generated world (2–3 template parts stiched together in random order);
- a playable world with some obstacles and a way to loose;
- minimal controls and abilities of player's character;
- a mechanism to end the game and display some statistics (e.g. distance travelled).

5.3.2 Stage II preview

In Stage II you can extend the game mechanics, add AI, multiplayer, advanced interface and more. Individual assignments may include:

- game menu;
- advanced game mechanics (e.g. bonuses, abilities, obstacles, enemies, time manipulation, etc.);
- multiplayer (local, network);
- virtual or augmented reality interface;
- a configurable artificial intelligence for mobs;
- client-server architecture.

5.4 Tower defense

Tower defense is a computer game genre where the player is building up a defense line to keep a continually incoming enemy army from destroying or invading a tower (or some other object).

In this project you need to implement some variation of a tower defense game.

Tower defense games are often tile-based with square or hexagonal grids to place defense structures on. Defense structures may have different shapes (and occupy multiple tiles) and different abilities.

Placing a structure may require some resources (credit, materials, time, etc.).

Enemies come in waves, with increasing difficulty. Usually, extra resources are given to the player after a successful defense for each wave, allowing to repair or restructure defense line before the next wave.

Figure 12: Plants vs. Zombies

Enemies usually come in straightforward, predictable paths and may have various abilities (e.g. they can be hard to kill or they can destroy defense structures).

The game is over when the tower is overwhelmed or sustained too much damage.

5.4.1 Minimal requirements

Minimal implementation should have

- 2–3 types of defense structures to use;
- 1–2 types of enemies;
- a single map with 2–3 paths for enemies to arrive;
- one (possibly predefined) wave;
- a mechanism to detect when a game is over and display statistics.

5.4.2 Stage II preview

In Stage II you can extend the game mechanics, add AI, multiplayer, advanced interface and more. Individual assignments may include:

- game menu;
- advanced game mechanics (e.g. more structures, enemies, maps, etc.);
- generation of levels (maps and resource allocation);
- level editor;

- multiplayer (local, network);
- reverse mode (send it enemies for someone else's tower defense);
- virtual or augmented reality interface;
- a configurable artificial intelligence for enemy waves and/or tower defense;
- client-server architecture.

6 Simulations

6.1 Fluid simulation

Smoothed particle hydrodynamics (SPH) is one the common methods for modelling fluids and gases. It is a quite simple from both the programming perspective and for understanding (e.g. compared to modelling on a grid). This methods was first proposed to model astrophysical phenomena, but is now used also for fluids, gases and deformations in solid bodies.

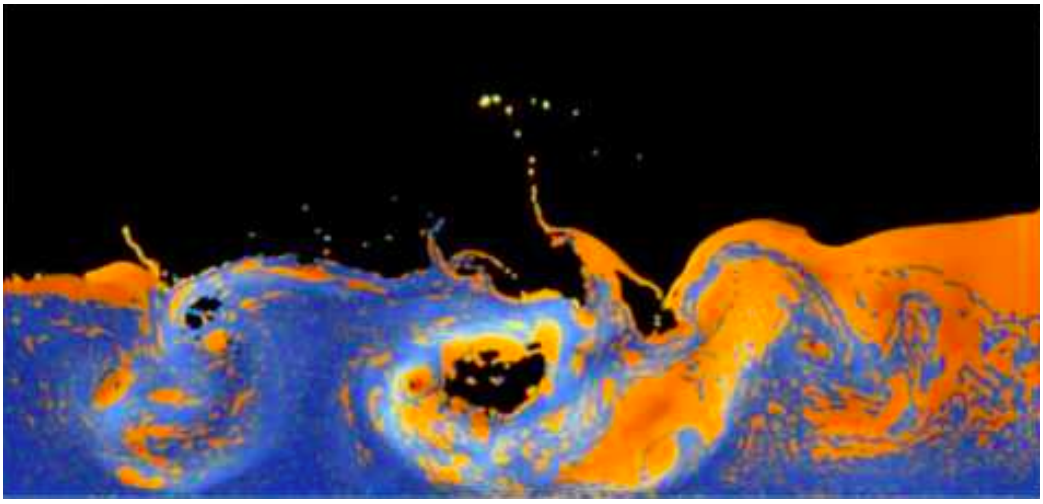


Figure 13: Modelling a fluid with 2 phases.

In SPH fluid is modelled as a collection of particles. Every particle actually represents a number of particles in the local area (i.e. not just one molecule) and has properties such as position $\mathbf{r}_i$, velocity $\mathbf{v}_i$, mass m_i. Values for this properties determine the properties of a fluid. Particles also have a "smoothing

length" h, which determines a distance at which the properties are "smoothed out".

The contribution each particle has at a given point $\mathbf{r}$ is determined by a so called kernel function $W(\mathbf{r} - \mathbf{r}_i, h)$. Most often Gauss function or polynomial splines are used for kernel. For the latter ones the value is always zero for particles located farther than h from $\mathbf{r}$. This allows for an effecient modelling, ignoring contribution from the far away particles.

Any physical value at $\mathbf{r}$ can be computed using this formula:

$$A(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h),$$

where A_j — is the value at $\mathbf{r}_j$, ρ_j — density of particles at $\mathbf{r}_j$. For instance, density can be computed as follows:

$$\rho_{\mathbf{r}} = \rho(\mathbf{r}) = \sum_j m_j W(\mathbf{r} - \mathbf{r}_j, h).$$

For particle movement SPH models pressure, viscosity and surface tension.

Pressure is computed from density:

$$p_i = k(\rho_i - \rho_0),$$

where ρ_0 — density of the environment, and k is stiffness coefficient. This parameter is not necessary, but allow for better fine-tuning. Pressure force can be computed as follows

$$f_{pressure}(\mathbf{r}) = \sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(\mathbf{r} - \mathbf{r}_j, h)$$

Viscosity is an effect that happens when fluid layers "rub" each other. Viscosity force depends on the velocity difference of neighbouring particles:

$$f_{viscosity}(\mathbf{r}) = \mu \sum_j m_j \frac{\mathbf{v}_i - \mathbf{v_j}}{\rho_j} \nabla^2 W(\mathbf{r} - \mathbf{r}_j, h),$$

where

$$\mu$$

is the viscosity coefficient.

Surface tension force can be computed using this formula:

$$f_{tension}(\mathbf{r}) = \sigma \sum_j m_j \frac{1}{\rho_j} \nabla^2 W(\mathbf{r} - \mathbf{r}_j, h),$$

where

$$\sigma$$

is the surface tension coefficient.

Different physical attributes can use different kernel functions to better capture neighbouring particles contribution. Most attributes are well modelled with W_{poly6}:

$$W_{poly6}(\mathbf{r}, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3, & 0 \le r \le h \\ 0, & r > h \end{cases}$$

$$\nabla W_{poly6}(\mathbf{r}, h) = -\frac{315}{64\pi h^9} \begin{cases} 6r(h^2 - r^2)^2, & 0 \le r \le h \\ 0, & r > h \end{cases}$$

$$\nabla^2 W_{poly6}(\mathbf{r}, h) = \frac{315}{64\pi h^9} \begin{cases} 6(h^2 - r^2)(4r^2 - (h^2 - r^2)), & 0 \le r \le h \\ 0, & r > h \end{cases}$$

However, for a better effect of uncompressible fluid (e.g. water) this kernel function is better suited for density:

$$W_{density}(\mathbf{r}, h) = \begin{cases} (1 - \frac{r}{h})^2, & 0 \le r \le h \\ 0, & r > h \end{cases}$$

So modelling is implemented with a step-by-step processing of a particle system. At every step for every particle you

- find the neighbouring particles;
- compute density;
- compute pressure;
- compute forces: pressure, viscosity, surface tension, gravitation, etc.;
- compute acceleration;
- update velocity and position (with boundary checks, e.g. walls);
- render.

6.1.1 Minimal requirements

Minimal implementation should have

- basic 2D visualisation and modelling;
- one scene with some shape containing the fluid;
- easily configurable model (via coefficients and kernel functions).

6.1.2 Stage II preview

In Stage II you can extend modelling techniques, add more types of particles, advanced interface, computation optimisations and more. Individual assignments may include:

- computation optimisations (e.g. efficient data structures for fast neighbor search);
- multi-phase fluids (multiple types of particles);
- 3D fluid simulation;
- complex environment (floating objects, watermills, etc.);
- a fluid-based logic puzzle;

7 Other ideas

You are not restricted to the project ideas above: you can propose a different project idea. However, you need an approval of principal instructor in order to start working on.

You can get inspiration in some computer science papers, blog posts and projects you are interested in!

Here a few rough ideas you can try to make into a project:

- a chat bot (e.g. Telegram bot);
- a blockchain system (a simple implementation in Haskell);
- a machine learning application (you can either implement ML from the ground up or try use an existing library);
- implementation of purely functional data structures or algorithms.