UPPSALA
UNIVERSITET

IT 14 059
Examensarbete 30 hp
Oktober 2014

The Steamroller Programming Language

David Forsgren Piuva

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

The Steamroller Programming Language

David Forsgren Piuva

During my research I haven't found any popular language with enough features for table driven programming to define the borders of the paradigm. The Steamroller programming language is trying to inspire new languages in a new direction with the beginners in mind using methods from human computer interaction. The resulting langauge is close to Basic (1964) and D (2001). The syntax is mostly inherited from Basic and purity levels are close to D.

The main feature in the written part of the language is table style enumerations. Graphical overview of projects makes it easy to prevent cyclic dependencies before they are created. Graphical code generating languages for data flow graphs and state machines are parts of the language as a whole to allow extremely complex systems to be displayed as something trivial that everyone can understand.

After having implemented and used the table feature, other imperative languages started to feel obsolete, repetitive and annoying to use because the table feature opened a new path to simpler and safer code that I did not see before.

Handledare: Lars Oestreicher
Ämnesgranskare: Olle Eriksson
Examinator: Olle Gällmo
IT 14 059
Tryckt av: Reprocentralen ITC

Table of content

1. Introduction 6
  1.1. Programming paradigms
  1.2. Development methods for usability 10
  1.3. Research questions 11
  1.4. Disposition
2.1. Scenarios with personas 12
  2.2. End user involvement 13
  2.3. Taxonomy
  2.4. Heuristic evaluation 14
3. Decisions and features of Steamroller 15
  3.1. Why a single file for everything? 16
  3.2. Why graphical project overview? 18
  3.3. Why tables? 20
4. Programming in Steamroller 22
  4.1 Graphical interaction
  4.2. Enumerations 29
  4.3. Modularity 33
  4.4. Methods 34
  4.5. Math
  4.6. Line breaks 46
  4.8. Identifiers 49
  4.9. Strings
  4.10. Types 50
  4.12. Control flow 52
  4.13. Using applications as functions 56
5. Modifying programming on the C level 57
6. Conclusion and discussion 58
References 59

1. Introduction

This thesis deals with the development of a new visual programming language, Steamroller, which is a programming language within the table/data driven paradigm. Most features and properties in the language are adopted from existing languages in order to put together the best features into something more usable.

1.1. Programming paradigms

In this section is a quick overview of the different programming languages and paradigms to show why the programming language Steamroller is needed for the data/table driven paradigm. All the other paradigms have successful languages that define the area of the most supported paradigm. Many programmers were however programming in the different paradigms as a state of mind long before languages came to help but with more mistakes from not having the safety features.

Discussions about paradigms and programming languages are usually so subjective and biased that programmers speaking about their favorite paradigm act like members of religious cults. It is not uncommon to see people on programming forums get scolded for blasphemy for recommending another paradigm.

Essentially, if you disagree with them then you just haven't learned how to use the paradigm properly. This is the programmer's version of "God's ways are unfathomable" with the addition of having transcended into understanding God's ways.

1.1.1. Traditional programming languages

The number of programming languages and dialects is very large, and it is of course not possible to describe all of them in a thesis like this. Therefore I have chosen to start by describing the various paradigms using a representative for each paradigm as an example of their properties.

Imperative programming (Assembler 1950) **(Kip R. Irvine, 2010)**
Imperative programming is essentially to program using side effects to assign values to variables. The program is written in longer threads where the code flow is mainly interrupted by the use of IF-THEN-ELSE and GOTO-statements. Most programming languages (with declarative languages as a major exception) have an imperative substructure within modules, procedures or objects.

Assembler is a group of programming languages for different hardware. Each statement in Assembler corresponds to an instruction or static memory in the hardware's machine code. A statement in Assembler usually start with a short name for the instruction to use.

Procedural programming (Algol 1958) **(Daniel D. McCracken, 1962)**
Imperative programming tends to produce programs with a confused structure and into something called spaghetti code in the long run. The term"spaghetti code" is used to describe the condition when the code becomes heavily entangled like a bowl of spaghetti because there is no clear division of sub tasks. Procedural programming was a first attempt to avoid spaghetti code by dividing code into reusable procedures with well-defined tasks to be performed.
What makes the code not procedural:

* Using goto statements.
  It can sometimes be necessary to use a goto statement
  but is not procedural nor structured programming.
* Having too much code in the same method.

Functional programming (Lisp 1958) **(Conrad Barski, 2010)**
Global assignment of values to variables introduces many side effects that can be difficult to understand. One attempt to avoid this was to introduce functional programming where functions and recursion are used and the programming has to be done without the use of global variables. This is a limitation which often becomes too big and this paradigm is often combined with other paradigms in order to be of practical use.
What makes the code not functional:

* Using side effects.
* Declaring global variables.
* Having too much code in the same function.

Object oriented programming (Simula 1962) **(Bjørn Kirkerud, 1989)**
To use abstraction and class inheritance to describe the behaviour of objects.
What makes the code not object oriented:
* A procedural application in one big class.
* A class without virtual methods.
* A class that only has public data and a constructor.
* A class with no data.

Modular programming (Pascal 1968) **(Larry R. Nyhoff & Sanford Leestma, 1988)**
To divide the code into modules so that reusable modules don't depend on non reusable modules.
What makes the code not modular:
* Exposing too many things in the global namespace.
* Having cyclic dependencies between modules.

Declarative programming (Prolog 1972) **(Ivan Bratko, 2000)**
To program by declaring relations and letting the language find a solution to the problem. This has the problem of not being deterministic.
What makes the code not declarative:
* Using side effects.

1.1.2. Special-purpose programming languages

Some application areas will have special needs from the programming languages. Sometimes these are not even considered to be programming languages. Some representative languages are listed here below.

Relational query programming (SQL 1974) **(Ben Forta, 2012)**
A paradigm specialized for searching in databases that has to be combined with a real programming language for everything else. Relational programming is an extension to table driven programming that is common in databases. The tables should contain indices to other tables to form relations and get a higher normal form. The relations are used to do advanced set operations.

1.1.3. Special hardware programming languages

Shader programming (Renderman 1986) **(Don Rudy Cortes & Saty Raghavachary, 2007)**
A paradigm that started in raytracers for the CPU and is now very close to the hardware of the current graphics cards to do functional calculations directly on the graphics card. It is mostly used for rendering triangle based 3D models but can also be used for generic problems like sorting an array, finding patterns in text, generating sounds or running physical simulations. With shader model 4, this must be done by representing all data as pixels in textures but compute shaders with shader model 5 are less restricted. Most of the performance comes from the paradigm of having each thread as it's own independent application so that there is no accidental communication between the threads. Abstracting away the order of execution also prevent some common mistakes when making vector field simulations (used for simulating wind and water) because the old state is clearly separated from the next state.

1.1.4. Table-driven programming **(Dave Thomas, 2005)**

Table-driven programming is essentially to replace IF and SWITCH statements with explicit tables or arrays where data affecting the control flow is stored. This is needed because hard-coded comparisons with constant integer keys becomes hard to maintain when the same comparinson is used on multiple places in different modules and suddenly you need a new key that should inherit the properties of a previous key with some minor changes. Then it is common to make mistakes and cause new bugs in the messy code.

To improve an IF statement's maintainability, store boolean flags for each key in a table and let the IF statement look up the flag from the table using the key. Tables can contain function pointers to implement a functional version of the strategy pattern when there is no need to contain global variables in objects. When the tables are getting too long and repetitive, try to normalize the tables into the "Boyce-Codd normal form" for more flexibility.

In the Steamroller programming language, there tables are a built-in feature to reduce the amount of code required to create and access a table. The feature also add more type safety because an enumeration contains the table and it is hard to accidentally use an index for the wrong table when the type of the index is used to access the table.

The border between "data-driven programming" and "table-driven programming" is not well established since most dictionaries contradict each other about these words. In general, data driven is more used to describe simple reading of properties or vector field computations while table driven is when the content of the table have a big effect on control flow. The problem is that most things will affect the control flow sooner or later. The term table driven is used for both in this document because it is programming using tables.

1.2. Development methods for usability

(David Beny, 2010)

The development of Steamroller has had usability as one of its guiding principles. The language should both be easy to use and easy to learn. In this chapter, the methods that I selected for developing the Steamroller programming language to make sure that the language focus on usability first will be described.

The use of personas:
Using personas is a method for requirement specification that involves the imagining of that you know someone who is the intended user and then try to predict what the person would do in many different situations to find usability problems.

One advantage of using personas is the ability to think ahead, maybe 50 years into the future when the latest operating system is forgotten legacy, the digital rights management system will lock itself and the insanely strong encryption can be broken instantly with a quantum processor. That is hard to do with real users unless you have a secret time machine. It can help to avoid stereotypes if you actually know someone like the persona but with different skills. User scenarios can also be mixed with an ethical analysis to get a bigger picture.

Establishing a taxonomy using cognitive tasks analysis:
A taxonomy is a structured description of a domain. The establishment of a taxonomy means to divide each task and object into the smallest parts and describe these in a semantic perspective. The motivation for this is to investigate how things can be done faster. Including delays from thinking is important because more time and energy is spent on thinking than on actually doing things.

Heuristic evaluation:
A heuristic evaluation is to use a set of defined measurements to detect usability problems. Although this is most often mentioned in the context of end-user applications, the idea can be applied also to the development of a programming language, such as Steamroller.
By evaluating one small part at a time, you can overcome your personal bias and see why other people dislike something that you like. If you don't dare to use subjective heuristics then the result will get very biased towards things with bad usability because human phychology is subjective.

1.3. Research questions

How can a programming language be implemented to define the table driven paradigm? This is relevant because someone wanting to learn a new paradigm needs a language supporting it well. There are tools that have been used previously for table driven programming but third party extensions that are not part of the language standard is not so convenient to use because they might not be reimplemented for each new compiler and then gets stuck in an old environment. After a while when the code generator has been abandoned, you will start to rewrite the code by hand to make it more readable and you are back where you started without the feature.

How can professional programming move closer to graphical programming without becoming too limited? This is an old problem that has held back the development of graphical programming to being merely for learning and visualization rather than being the final programming tool. There is no ideal solution to the problem since simplicity often comes at the cost of not being able to do everything. All I can do is to find a balance between toy language and redundant code.

1.4. Disposition

2. The development process
How the langauge and integrated development environment was developed for usability using personas, heuristics and taxonomy.

3. The main features of the language
The pros and cons of the main decisions.

4. Documentation for the language
In depth documentation about the language.

5. Extending with C
What you should know before inserting C code into your steamroller project.

6. Conclusion discussion and future development
Where the language can go next.

2.1. Project maintenance scenarios with personas

The use of personas to describe the different types of users is a method that has been used mostly for software engineering, but it is also useful for the development of programming languages. In this work I have used minimised personas, that do not cover many of the purely personal aspects. Personal aspects can make the personas more living but comes with the risk of excluding a large group of real users.

2.1.1. Experienced programmer persona

Anna, 42, is an experienced programmer who has developed computer programs in different languages for a long time. She has several private projects on the side of her work, and is eager to learn new ways of programming. She has also used many different languages, and has a good knowledge about the different paradigms, although she prefers using a well-defined programming environment, such as Microsoft.NET. In her daily work she is responsible for programming of special modules, which require high proficiency.

With the knowledge about database normalization, she can produce compact code of high quality that most programmers can understand. However, sometimes she finds it difficult to convey the basic ideas in her more advanced procedures, which makes her frustrated and sometimes even causes her to reconsider her choice of work career. If only there were a better way to describe the ideas in the code.

2.1.2. Intermediate programmer persona

John, 34, is an intermediate programmer who reads a lot of other people's code, debugging it and suggests improvements. He also writes his own code and prefers to do this in Java, although he has some knowledge about functional programming. During his studies he came across Alice, which is a limited object-oriented graphic programming system, which although it was on a toy level of complexity did give him some ideas about using graphic programming tools. However, he still thinks it is important to have a well structured "real" programming language to work with.

He has found out that reading both his own and other people's code is much more common than writing code. The language's syntax is often close to spoken languages in statements.

```
Sub Foo(Val N As Int)
    Push N To MyList
    Resize MyArray To 1..N
End Sub
```

However, he has also found out that when programming languages are too close to spoken languages in expressions, especially in combination with negations and boolean logic, this often causes ambiguity and confusion since it is not natural to mix parentheses with spoken words when different people parse spoken grammar in different ways

Example of such an ambiguity when nesting in spoken language:

All cars are not yellow.

(All (cars)) are (not (yellow))

A computer would most likely interpret it as "No car is yellow."

Not((all (cars)) are (yellow))

A user would most likely interpret it as "There is a car that is not yellow."

John has thought hard about how to explain the negations in a better and more consistent way, but has not found a good way to handle this.

2.1.3. **Beginner** programmer persona

Anders is still learning how to program in a large scale. As a beginner programmer, he is not aware of how many bugs that are hiding in the code. This is mainly because he is not experienced in software testing. He has problems reading the code, even his own after some time.

2.1.4. Customer

Elisabeth, 30, is a customer of the product developed by the company. She is especially interested in changing some parts of the program, though she is not interested in programming at all. She would like to have the ability to change the program's behaviour by just making simple adjustments, e.g. as setting properties in a table. The more that can be done by the users themselves the less need there is for having programmers responsible for those small changes in the behaviour.

2.2. End user involvement

It would be great if users could be involved during the development but that is not the case in this kind of project. To teach a beginner programmer how to program in a language that is constantly changing and not used in any workplace would be both difficult and unethical. The time budget would be exceeded and most of the feedback would be things that programmers have already written about when asking for help with already existing languages on internet forums.

(Constantine & Lockwood, 1999) describes the classic methods for working with usability when you can't use a test subject for some reason. It can be a complete disaster if you don't have enough knowledge about what people actually want but you can do things that otherwise would take too much time to develop.

2.3. Taxonomy

A lot of time was spent on the taxonomy of **refactoring** from naive implementations into tables of high normal form. The result is a table extension to the enumeration that give some of the power from object orientation while keeping the power of an index.
The taxonomy of **modularity** were also investigated to make sure that moving methods between different modules is easy. An idea about complex interfaces had to be abandoned because moving a method from an interface would require each caller to change like in object orientation but without the benefits.

2.4. Heuristic evaluation

A version of heuristic evaluation was used while keeping a list of the language's biggest weaknesses in order to identify and try to fix the largest weaknesses of the language during development. The heuristics that were developed in this work are some of the most commonly discussed guidelines found in debates about which language is better.

1. Avoiding global state to increase code reusability
In Java it is possible to instanciate the global state but this does not remove the problem if the whole application accesses the same main object for all global variables.
Steamroller supports functional programming for large amounts of reusable code.

2. Determinism and safety
Both determinism and safety are fundamental properties, in order to facilitate rapid application development because a language that works differently on different computers and which is not memory safe can not use testing to measure the quality of the code. Pointer errors may randomly crash the system while still giving an output that is according to the postcondition.

3. Low level interface access
A language that can't access low level interfaces by itself will be depending on someone else making components with limited access to the new hardware features. The Steamroller programming language allows for the insertion of pure C code directly into wrapper methods so that the programmer can use all the dangerous performance optimizations needed to take full advantage of the system without having to write the whole project in a low-level language.

4. Speed
People who say that performance does not matter are usually also the ones ending up doing the most work on optimizing because they used interpreted languages without using the advantages that they offer. The faster your languages and engines are, the less time you have to waste on optimizing performance when realtime deadlines are broken. The Steamroller programming language only have a reference implmentation that in the current state still wastes some memory allocations from possible memory aliasing but integer and float math is however very fast by compiling to C.

5. Simple dependency
That dependency is a malicious property in programming is known by programmers who have programmed long enough to see the old applications malfunction due to an obsoleted third-party component which is no longer supported. Steamroller only uses standard C libraries and can replace the backend C compiler for compiling to other operating systems than Microsoft Windows.

6. Prevent bad patterns
Bad program structures are difficult to change, and if bad design patterns are available, these will be more harmful to beginner programmers and people who are switching, between different languages, but not extremely skilled on one. This is a common argument against programming languages with weak type safety like Python and Basic. Method purity levels in the D language and Steamroller allow detecting accidental use of global variables after duplicating code into a new function. The reason for the low score for C++ is that there are so many ways to do the same thing that there are many pitfalls in modularity and cooperation. Visual Basic 6 scored higher than Java because Visual Basic can enable strictness by writing "option explicit" as every beginner learns from a good book while Java have outlawed functional programming so that you are forced into bad patterns like instanciating the math library.

7. Detect errors in compile time instead of run time
Early error detection is important for producing high quality applications that are correct and do not randomly crash at the end-user. The Steamroller programming language uses static allocation of all collection heads to eliminate the famous "null pointer exception" completely. Only dynamic size allocations may use dynamic heap allocation.

8. Prevent unwanted module dependency
The graphical overview in Steamroller allows for a better planning by the programmer before the code gets cyclic dependencies.

2.4.1. How different languages pass the heuristics

Steamroller will of course have a big advantage in this test because these heuristics were used to design the language. No language can be good at both 2 and 3 because they contradict each other.

	1.	**2.**	**3.**	**4.**	**5.**	**6.**	**7.**	**8.**
C	Bad	Bad	Good	Good	Good	Bad	Good	Okay
C++	Okay	Bad	Good	Good	Good	Awful	Good	Okay
Java	Okay	Okay	Okay	Good	Bad	Bad	Good	Okay
ML	Good	Good	Bad	Okay	Good	Good	Good	Okay
Python	Good	Good	Bad	Bad	Bad	Bad	Bad	Okay
Visual Basic 6	Bad	Good	Bad	Good	Bad	Okay	Okay	Bad
Steamroller	Good	Okay	Okay	Good	Good	Good	Good	Good

3. Decisions and features of Steamroller

This chapter explains the reasons behind the 3 most important design decisions in Steamroller.
The name "Steamroller" comes from the normalization into tables that make the code flat.

The following features are characteristics for the language implementation:

- **3.1.** A solution of projects is stored within a single file.
- **3.2.** A graph is used for module overview.
  - Graphs are also for code generation with data flow graphs and state machines.
- **3.3.** Tables are built-in to the written core language.

3.1. Why a single file for everything?

When I first learned Visual Basic 6 at the age of 11, I was confused by the complex way of saving projects with many different files. I did not know which files I could delete and which files were an important part of the project. This feels like a legacy remaining from console line compilers in DOS from the time when the work memory could not store all the source code at once.

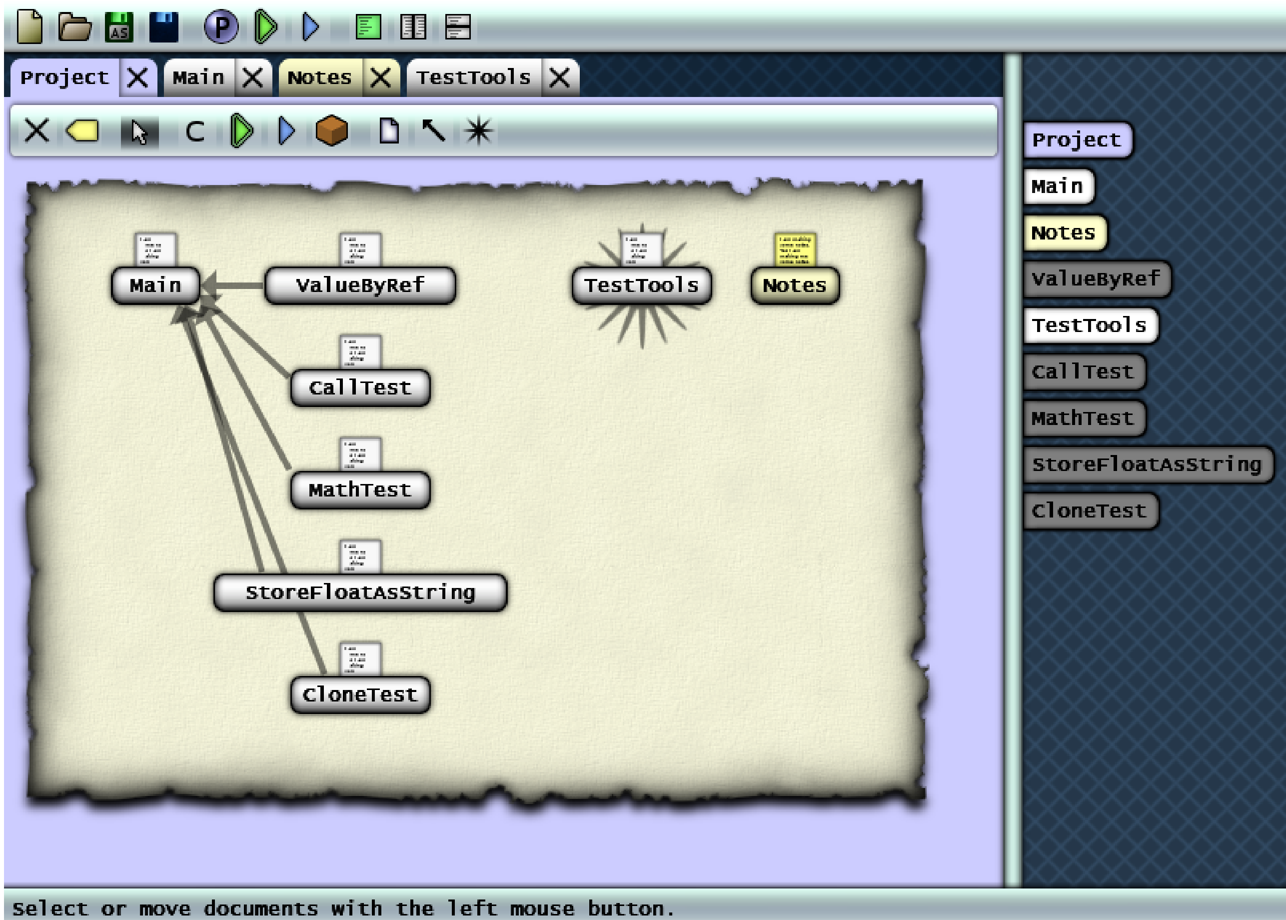
Problems:

* Missing source files when using code backups or sending the code to another programmer. This is one of the worst problems when sharing C++ code because there are always some libraries that are required. Sometimes there are however juridical and practical reasons for not giving out the source code with everything included. A library might be very large or have an exclusive license where the license number can not be distributed. We can however solve the problem with small source files that are just forgotten and link using the C programming language for a large library. Dynamic linking is preferred in Steamroller because there is no standard syntax for static linking of libraries in C99. Another problem is that most dynamically linked libraries for C also require a static library for calling the dynamic library that has to be rewritten in Steamroller.

* Cluttered file system. This make system maintenance much slower than having a few large files. If a file of source code only need 400 bytes but the padding size of the file system is 4000 bytes then a lot of space is wasted for no reason. Computer games often use files that contains archives of many small files to reduce disk fragmentation, save space, allow a weak encryption to make it harder to steal artwork and simplify checksum verification to prevent cheating and broken installations. A strong encryption on packet files is impossible and that is why many digital right management systems can't protect a popular game against pirates for more than 24 hours.

Solution:

Everything that is compiled together is stored in a single solution file. Each document in the solution file have a unique ID for referencing to prevent duplication of document names. This simplifies the way of thinking about the projects when everything needed for deterministic compilation is included in the same file.



***3.1.1** The visual user interface*

Advantages:
* No more missing source files because you either have the file or you don't.
* Allow storing many modules without wasting space on padding in the file system.

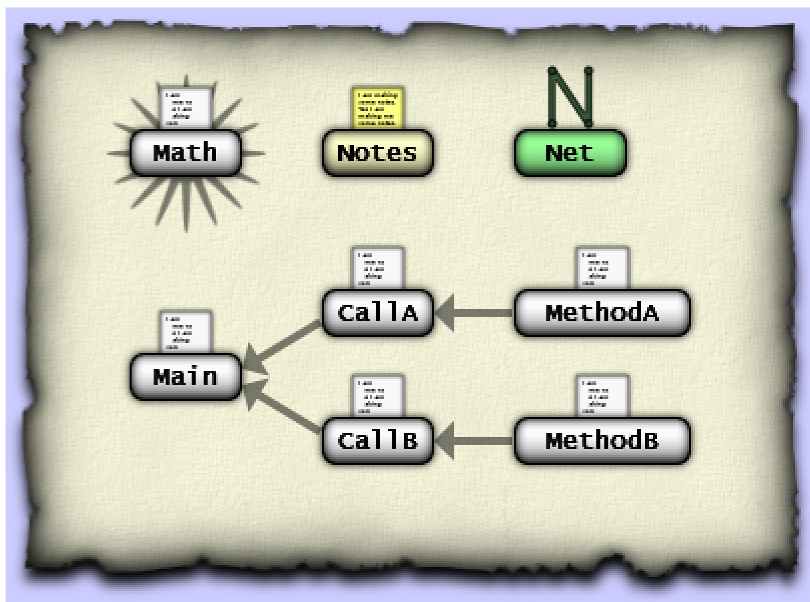
Disadvantages:
* Some users might feel insecure about letting one file store everything even if the database system is very lightweight and easy to understand.
* Existing version control systems will not work on the database file if there are conflicting changes to the same solution. They must be custom-made for the database system.

3.2. Why graphical project overview?

Large projects need many modules but a plain list of modules do not use human memory in an efficient way because the location of something is relative to the location of the perceived context. If I move the list of modules from the right side to the left side in Visual Studio, I will keep looking for modules on the right sides for months until I got used to the new X coordinate. If I remember their locations in two dimensions then I might as well take advantage of that and create a graphical representation of the mental model that is used to deal with the complex system. There is a lot of useful knowledge about human spatial memory from the Japanese card game Karuta (かるた) because the Karuta players must also pick one of many things quickly.

Problems to solve:
Late overview of dependencies between modules leads to cycles of dependencies that can be seen as a "yarn ball" programming pattern because no module can be removed no matter how many classes it use to be reusable in theory.

Solution:



3.2.1 *The graphical project overview showing how modules are connected*

Drag arrows between modules to make them visible to each other. Enable a "sun" visibility to make a functional module visible to all other modules. The module visibility graph decide which documents from the solution file that the project will use. This way you can see how the modules can access each other with possible dependencies. The goal with seeing this graph is to actively think early about how the number of dependencies can be reduced between the reusable and the non reusable modules.

Don't get religious about modularity because it makes no sense to reduce dependency more than needed between modules that are not going to be reused for any other project. As long as everyone in the project can handle the complexity, it is okay to break the encapsulation of another module when lots of performance can be saved by doing so. The graphical user interface for the Steamroller programming language could not have been nearly as good with generic visual components because the graphs require special rendering and the text editing must be custom made to make the indentation and color coding exactly as specified by the langauge.

It is good practice to have extra project in the same solution for module testing with alternative versions of the surrounding modules that tries to simulate the worst case scenarios for the tested module. For example, one might want to test a lossless compression algorithm used in the application but don't want to destroy the original application with a heavy regression tests running in the beginning. Having tests in the final application can also prevent optimizations in the C compiler and cause major performance issues.

Advantages:
Makes it easy to check that reusable modules don't depend on non reusable modules.
Easy to remember the locations when using your own mental model and absolute 2D positions.

Disadvantages:
It is hard to agree on a standard file format for graphs that everyone can trust their work with.

Alternatives:
The same control is given with the header system in C/C++ but not with the same overview.
I thought about letting the programmer draw unique icons for modules but programming can be quite abstract and drawing the icon would take time from the real work.

3.3. Why tables as a built-in language feature?

When I make a computer game, I often need a collection of data to store the types of buildings, vehicles, monsters and other things. These are then stored in arrays, lists or functions with a lot of extra code, side effects and bad overview. Some of these collections will eventually be stored in files to allow making expansion packs and mods (third-party game modifications).

Problems to solve:
A big amount of code is just hard-coded data for different options. Especially in switch cases.

Solution:
Extending enumerations into tables. A dot notation for getting properties from an index of the enumeration's type make the access more safe and less redundant. The table columns in an enumeration are a write protected fixed size array within the enumeration's namespace. The array's type is generated automatically from the datatypes of the columns.

```
Enum Color
    Key          | R As Int | G As Int | B As Int | Name As Char List
    Black,None   | 0        | 0        | 0        | "Black"
    Red          | 255      | 0        | 0        | "Red"
    Green        | 0        | 255      | 0        | "Green"
    Blue         | 0        | 0        | 255      | "Blue"
    White        | 255      | 255      | 255      | "White"
    Gray         | 128      | 128      | 128      | "Gray"
End Enum

Enum Item
    Key     | Sharp As Bool | Hard As Bool | C As Color  | Action As Event
    Arrow   | True          | True         | Color.Black | Shoot
    Stone   | False         | True         | Color.Gray  | Pick
    Pillow  | False         | False        | Color.Blue  | Sleep
End Enum
```

3.3.1 Enumerations with a table extension as shown in the source code

In this example, the enumeration called "Color" contains the columns "R", "G", "B" and "Name" that are stored in the array. To access the amount of green light in the white color, you can write either "Color.Table[Color.White].G" the old-fashioned way or just "Color.White.G" for extra safety and readability. The explicit access to the table exists in case that the programmer need to take out every property of a key at once without repeating the attribute name for every member.
If you declared a variable called "MyColor" with the Color enumeration as the type then you can skip the namespace and just write "MyColor.G".

The enumeration called "Item" is using "Color" as a datatype for referencing to the color table. This allow normalizing a table with indices to other tables. Just write "Item.Stone.C.Name" to get the name of the stone's color ("Gray"). The property called "Action" has the name of a routine as it's datatype to store write protected method pointers connected to the items. Method pointers assigned to a method pointer variable must be compatible with the abstract template method used as the type. An abstract template method is just a method that has a default behaviour or an error message.
Write "Call(Item.Arrow.Action)()" to call the arrow's action without any input arguments.
Call() takes a method pointer and makes a call using it.
"Call(MyMethod)(1, 2, 3)" is the same as "MyMethod(1, 2, 3)".

Advantages:
* Direct overview of the raw information that affect the application's behaviour.
* Cleaner syntax than using raw arrays while having the same benefits.
  "Cars[People[People[Me].Father].Car]" is written as "Me.Father.Car" for "My father's car".
* Compile time type safety against using an index with the wrong table.
* Separation between the large amount of constantly changing information and the reusable code.
* Leads to a flat design in a high normal form that is easy to serialize when something has to be saved to a file or computed on the graphics card. Using cycles of pointers is a recipe for disaster if you ever want to save the data to a file.

Disadvantages:
* Anything that is not in the first normal form can cause variable length cells in the table and waste a lot of **horizontal space**.
  This require the programmer to have basic skills in normalization.
* Inserting too many function pointers to the tables might eventually create a need for **class objects** when there are too many variables that are only used in some cases.
* Having function pointers in tables remove the ability to easily refactor the table into a database since function pointers can not be **saved**. This can however be solved by normalizing the function pointers into their own table that will not be stored in a database.

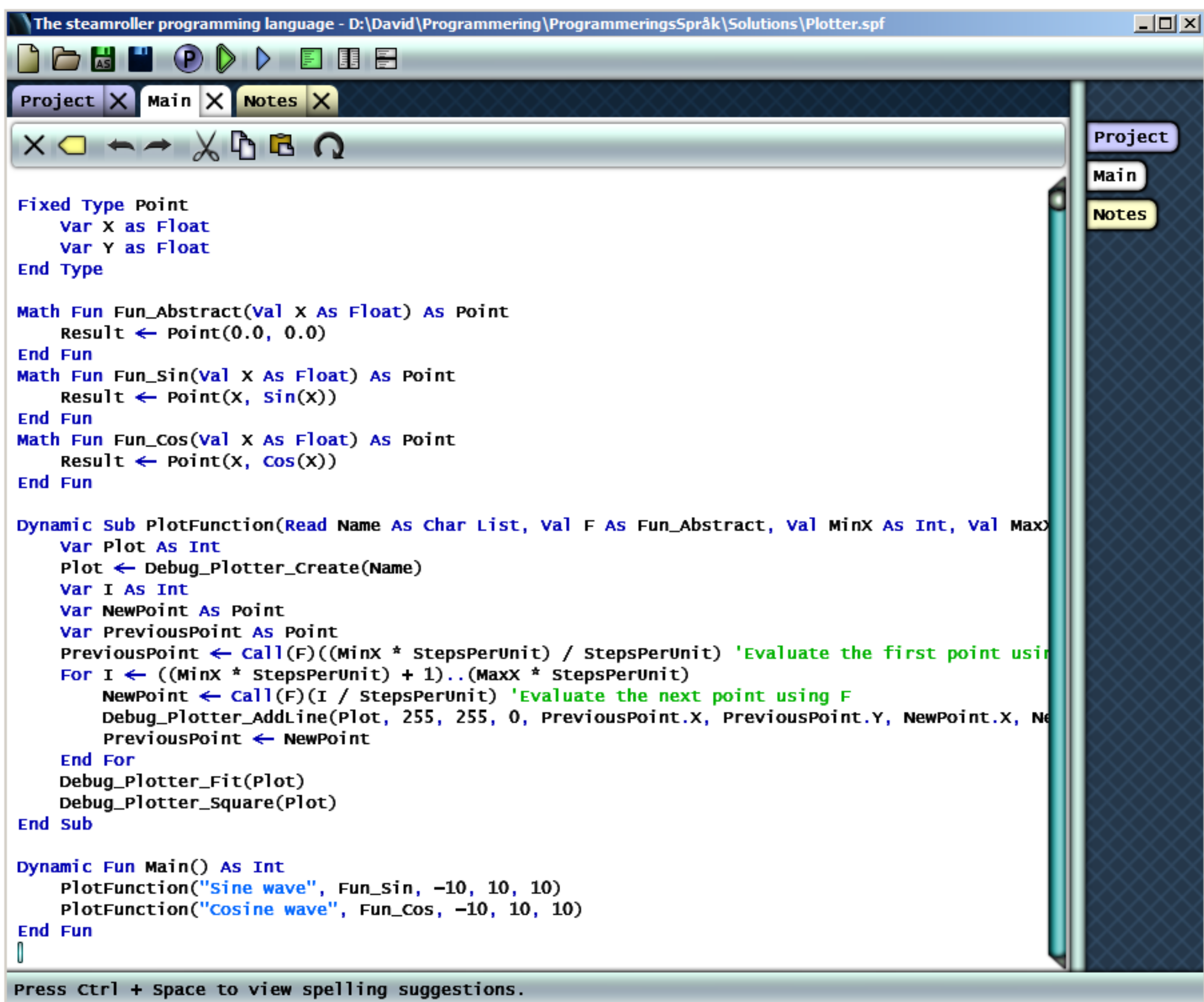
Alternatives:
* Enumerations with functions
  + Lightweight
  + Can compress the information when it is repetitive or can be calculated.
  + It is possible to make it type safe in some languages by taking an enumerated or tagged integer as the argument.
  - A single typo can cause the whole property to return the wrong value for each row.
* Enumerations with arrays written to using side effects
  + Can use a template for initializing each row and only write the differences.
  - Require initilizing the table before anyone may use the content.
* Enumerations with arrays filled at their declarations
  + Easy to duplicate a row and inherit all it's properties.
  - Wasting space for properties that are only true for one row.
* Loading everything from files at once
  + This is eventually the goal with most tables and might save time in some cases.
  - Overkill for small tables since version conflicts and missing files must be handled.
* Code generation
  - Accessing the data is still like a raw array without type safety for the index.
  - The programmer will not bother with external tools for 10 lines of code.
* Class inheritance
  + Can be extended into real objects if needed.
  - Fragmenting the memory.
  - Expensive refactoring from enumerations.
  - Hard to serialize into a file.

4. Programming in Steamroller

An introduction to how it is to program in Steamroller. This should be easy if you already know one of the Basic languages from before year 2000. The main idea is to keep simple things as compact as possible without being too cryptic. The language will not force patterns of over engineering that only have benefits in multi million line projects since most applications are just basic algorithms for computing something that the calculator and spreadsheet can't do. The interface must be compact and fast enough to feel like a small text note for something temporary. Most of the design decisions in this chapter are a matter of personal taste and can therefore not be given as strong arguments as the main features.

4.1 Graphical interaction

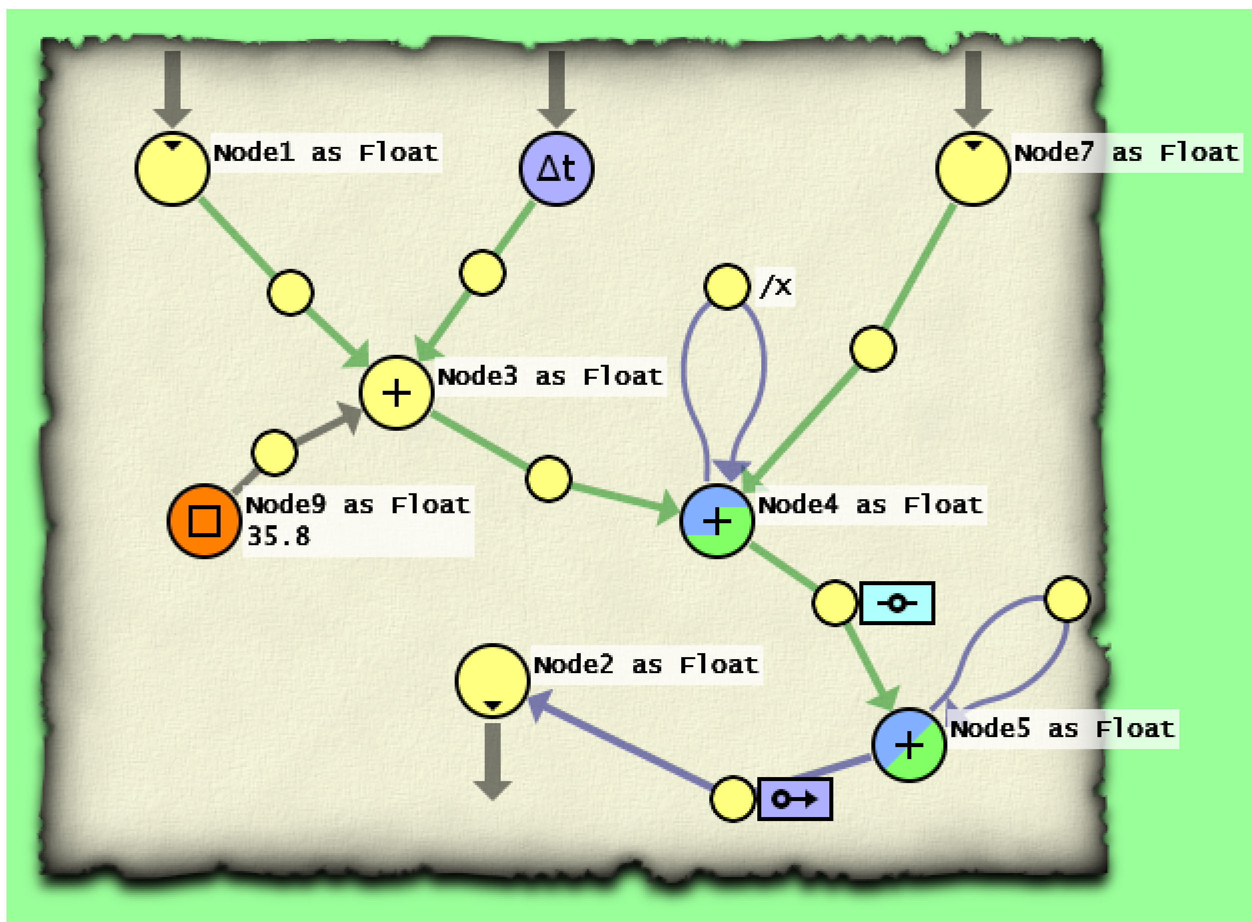
A big part of Steamroller is being a more graphical language without sacrificing power or usability. This is what the integrated development environment looks like when writing code.



***4.1.0.1** The integrated development environment*

4.1.1. How data flow graphs are used to generate functions

Each data flow graph contains both write protected code and a graph for generating the code. Each node is a variable that store the result of a calculation. Each arrow link indicate which node to read the result from as an input to the next node. Nodes have their values evaluated from top to bottom or left to right if they are at the same height. A link can be told to sample the value from the previous step and automatically force the node and the whole generated method to remember the provious value. If a link read a node's value using the midpoint method then a new input argument is generated where a copy of the state half a time step into the future is expected. Trying to read the new value before it has been written will give a warning since it is a very common mistake that is hard to detect without a graph.
To add 2 numbers, create a sum node and drag arrows from the 2 nodes that outputs the values that you want to add. Then you can drag an arrow from the sum node to use as input for other functions.
To make a division, create a product node and apply a reciprocal function to one of the inputs on the arrow. The same is done for subtraction using sum and negation for symmetry and simplicity.
To make a timer, create a continuous increase node that takes a node with the desired speed as input. Increase nodes can be combined with different functions to increase with the sum or product of all it's inputs. A continuous increase will automatically multiply the change with the time step and create a new input argument for time step if needed. If you use fixed timesteps then you can save performance with a discrete increase node that just adds it's function's output to it's previous value.



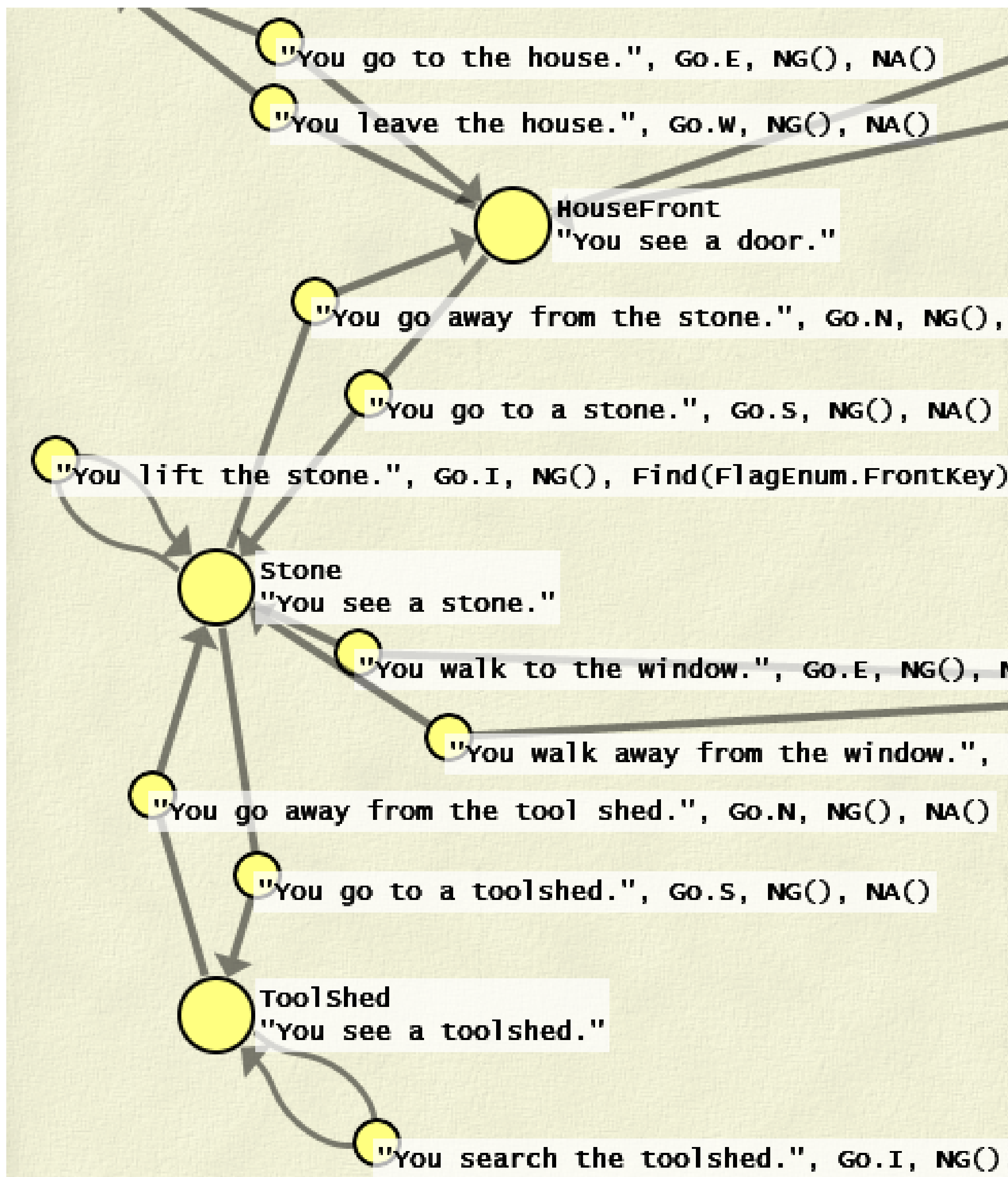
4.1.1.1 *A data flow graph*

The data flow graph allows calling functions declared in other modules by using a custom expression as a node or link's function. The custom expression is pasted directly into the generated code with all safety done in the written language to make it more powerful to use. You can also make nodes using your own datatypes defined in other modules. Just remember that everything accessed from other modules must be made visible to the module in the project's visibility graph by drawing an arrow to the graph's module or making a sun of the module to show everywhere.

The most powerful use of data flow graphs is the ability to create cycles to solve complex differential equations using simulation. When reading the value of a node, the default behaviour is to give a warning if trying to read it's output before it has been generated. When a cycle is needed then we must read from before the value has been written somewhere where it will have a minimal effect on the result. An euler forward link will just take the old value of the input node and hope that it is close enough to be good. A midpoint link will read half a time step into the future from an alternative state given as input to the generated method so that we can approximate the average value over the whole time represented by the step. When calling the generated method, you can start by running the simulation with the previous state as both current and midpoint inputs and let time delta be half of the actual time step to generate a midpoint step. You can repeat this process to generate an even better midpoint approximation unless it becomes unstable. Then you give the previous state and the final midpoint state as input to a call with time delta as the full time step to get the next state of the system.

4.1.2. How state machines are used to generate state tables

Each node is a state and each link is a transition. A table of states and a table of transitions are generated from the graph with information about how to follow the transitions to other states. Since there are practically no limit to how many ways a state machine can be interpreted, the graph will only generate the 2 tables filled with datatypes to be defined in another module. Each of the datatypes are to be used as a bag of properties where the programmer can enter all the commands, guards and actions needed for the type of state machine. It is recommended to let guards and actions have complex types containing a function pointer and input arguments of reusable types so that you can have a constructor written for each call as if you made the call with varying numbers of input arguments. Then the loop that interpret the tables will make calls using the function pointers.

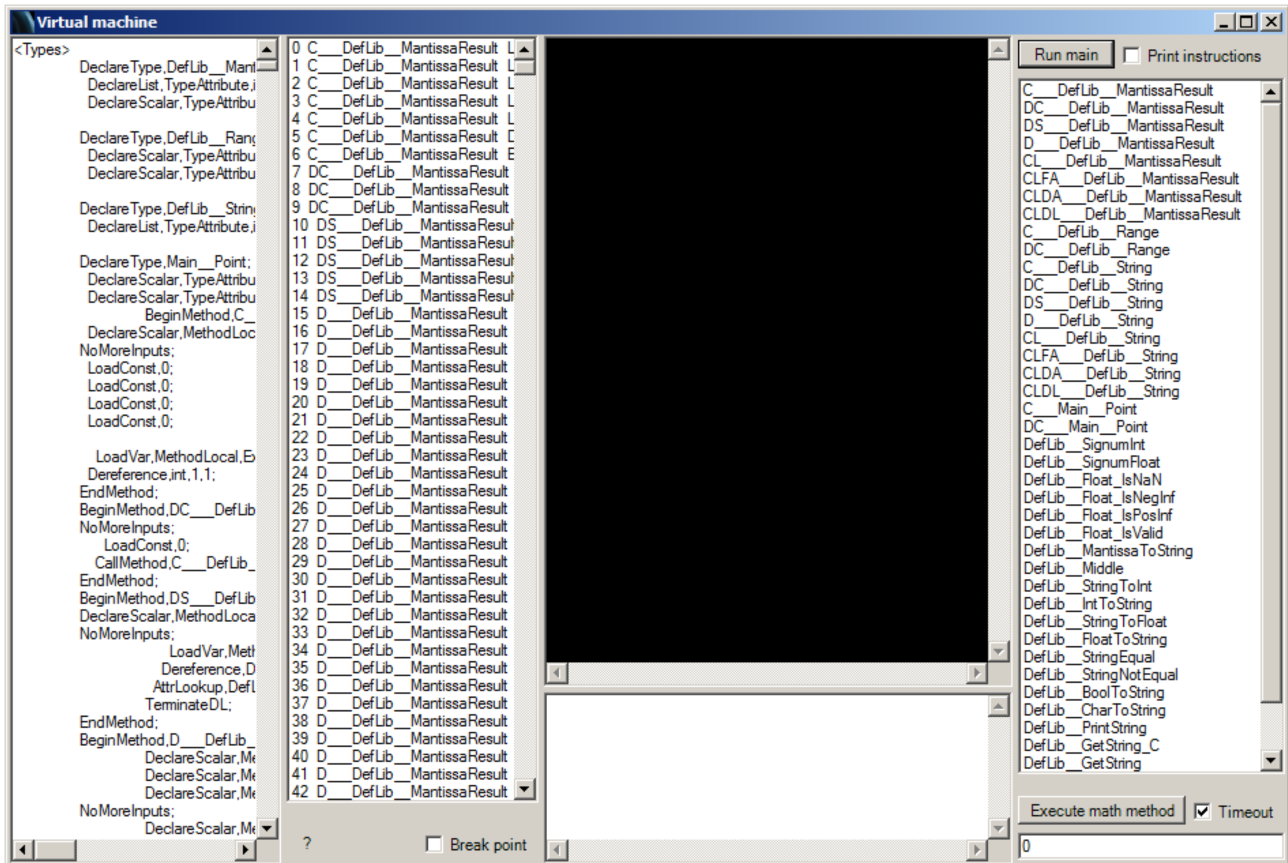


4.1.2.1 *A state machine for a text based adventure game*

To use a state machine, you need to fill two datatypes for states and transitions with the variables that you want to store in them. The state's type should contain the information that you need about what to do when reaching, being at or leaving a certain state. The transition's type should contain a representation of the input that may trigger it, guards that can prevent it from being used and any action to do when the transtion is being made. For method pointers, it is recommended to have a datatype that contains the method pointer and the input arguments so that compact constructors can generate the data needed to make a call without inserting a pile of empty arguments. Each state is a key in the generated state table and every transition is a key in the generated transition table. Your own data is inserted as a property in each of the tables. Other columns contains data about how the states and transitions are connected in a way that makes it fast to execute the state machine. To extend the state machine with variables, make a datatype that contains everything that belongs to the state machine that can be given as a context by reference to the abstract action method. This is object orientation in a functional syntax close to how it works in C99 and Python. The advantage of functional style object orientation is that you will know exactly how classes works under the surface without having to learn about how each C++ compiler generate pointers to virtual function tables and all that mess. This is useful when you need to optimize memory recycling, align the memory layout to reduce trashing of cache or make an interface to another language. The downside is that you will be repeating a lot of context pointers and won't get spelling suggestions for methods connected to the data.

4.1.3. The virtual machine

To have a well-defined reference implementation instead of relying on odd behaviours from different C compilers, the Steamroller programming language is compiled to both C and a virtual machine's assembler code (VMAC).



4.1.3.1 The virtual machine

Making a real debugger would require a few more years of development but it is at least possible to set breakpoints in virtual machine code for finding bugs in the virtual machine itself.

The benefits of using a virtual machine:

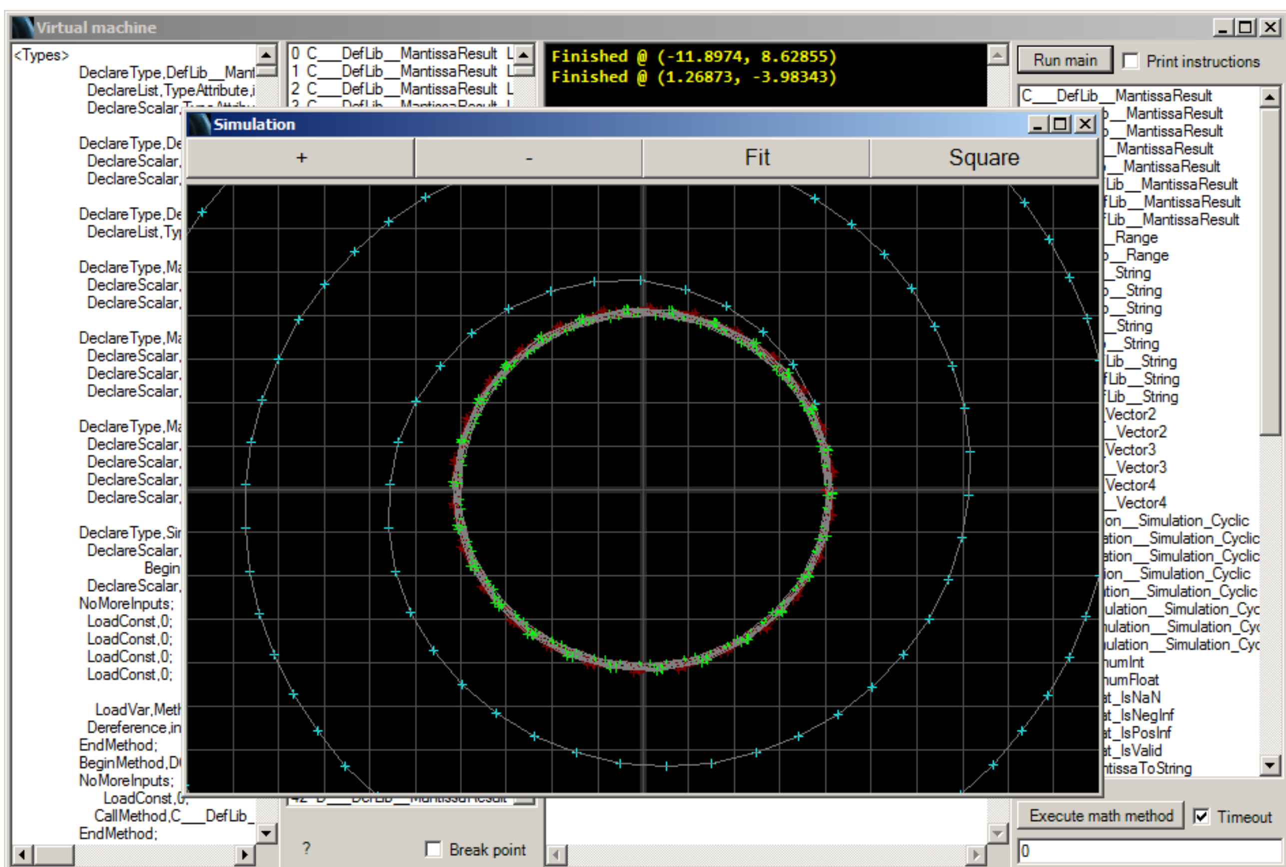
It is much easier to implement a debugger for a virtual machine since high level data strucutres can exist at the lowest level. This virtual machine has dynamic memory allocation and lists built-in to it's virtual hardware. If something goes wrong on assembler level then the machine is so simple that it can be fixed in minutes. Most often, the machine will catch those errors early with safety checks. All applications running in it are sandboxed by default when all calls to the operating systems must be handled by the virtual machine. This allow safe execution of code written by someone you don't trust to change things on your computer.

Code that is exclusive for a virtual machine can easily access system calls for scripting within another application. This allow downloading a plug-in from the internet without even knowing who made it because there are no dangerous system calls connected to the virtual machine and the application can place a wrapper that asks for permissions if something dangerous must be included. The safety problem with scripts on websites is that developers keeps asking for more powerful calls until the scripts are almost as dangerous as native machine code.

The problems with using a virtual machine:
Extending the framework takes a lot more time and there is a division between pure Steamroller code and code that have inserted C code to access modern libraries.
Debuggers has to be implemented for native execution anyway when more performance is needed just to run an application or it depends too much on a C library.

4.1.4. Plotter

The virtual machine is also a tool for testing pure math and have a plotter tool built-in. The plotter can only be accessed when running on the virtual machine because the extra dependency would make the final application tied up to visual operating systems for no reason.



4.1.4.1 The plotter executed by the virtual machine

The plotter is mostly useful for showing the results of a simulation and a convenient way to explore math and side effects. Create a new plotter window and allocate points and lines as vector graphics to use the plotter. Vector graphics allows zooming in on dense areas to see things that a raster image would not be able to preserve. This is because vector graphics store the positions for a collection of objects instead of a two-dimensional array of colors.

4.2. Enumerations

Enumrations are integers with extra type safety and their own namespace of named constants. Define multiple keys on the same line separated by a comma "," to let all the names get the same value. This is used to count keys and declare intervals of keys that are iterated over in for loops. Actually counting the total number of rows would be a global state that prevent safe refactoring of large scale systems.

Grammar:

```
Enum <Name>
    <KeyName> ( , <AliasName> )*
    ...
End Enum
```

Example:

```
Enum Shape
    None, First
    Box, FirstShape
    Sphere
    Cylinder
    Cone, Last, LastShape
End Enum
```

Shape.None and Shape.First are constant integers returning 0.
Shape.Box and Shape.FirstShape are on the next line and will therefore return 1. Shape.Cone, Shape.Last and Shape.LastShape returns 4 because there are 5 lines with keys and counting starts from 0.

4.2.1 Tables

Tables are an extension to the enumeration instead of a completely separate language feature because programmers already know that an enumeration is an integer and that integers can be saved to files. If the table syntax become widely used in the future then this design decision will no longer be preferred because of the ambiguity that is created between simple enumerations and table style enumerations. Adding properties to an enumeration is an easy step of refactoring and will keep the code very compact.

```
Enum Color
    Key           R As Int   G As Int   B As Int   Name As Char List
    Black,None    0          0          0          "Black"
    Red           255        0          0          "Red"
    Green         0          255        0          "Green"
    Blue          0          0          255        "Blue"
    White         255        255        255        "White"
    Gray          128        128        128        "Gray"
End Enum

Enum Item
    Key       Sharp As Bool   Hard As Bool   C As Color    Action As Event
    Arrow     True            True           Color.Black   Shoot
    Stone     False           True           Color.Gray    Pick
    Pillow    False           False          Color.Blue    Sleep
End Enum
```

4.2.1.1 Enumerations with a table extension as shown in the source code

The tables are an important aid in refactoring code before reading the data from a relational database because then the programmer have already done the hardest part of separating data from logic. Just copy a row in the table to inherit all the properties from the old row unless the programmer is making hard-coded comparisons with the key. The compact syntax of the table makes sure that the programmer have a good overview of all values of a certain property. The index to the table is an enumerated integer that can easily be saved to files without having to serialize class objects. A variable of the table style enumeration is an interface to the final table and can access the properties related to the key stored by the variable.

4.2.2. How to use tables

```
'Very bad code
Var IsCar As Bool
Var IsHouse As Bool
Sub Move(Read Offset As Vector3)
	If IsCar Then
		Location ← Location + Offset
	End If
End Sub

'Bad code
Enum TypeOfItem
	Car
	House
End Enum
Var Item As TypeOfItem
Sub Move(Read Offset As Vector3)
	If Item = TypeOfItem.Car Then
		Location ← Location + Offset
	End If
End Sub
```

Start by refactoring into enumerations

When you start to have too many mutually exclusive boolean flags, combine them into an enumeration to reduce the input space, remove preconditions and allow exhaustive testing. These code example are made to be compact and don't show how code is divided into multiple modules.

```
'Okay code
Enum TypeOfItem
	Car
	House
End Enum
Math Fun CanMove(Val Item As TypeOfItem)
	Result ← Item = TypeOfItem.Car
End Fun
Var Item As TypeOfItem
Sub Move(Read Offset As Vector3)
	If CanMove(Item) Then
		Location ← Location + Offset
	End If
End Sub

'Good code
Enum TypeOfItem
	Key        | CanMove As Bool
	Car        | True
	House      | False
End Enum
Var Item As TypeOfItem
Sub Move(Read Offset As Vector3)
	If Item.CanMove Then
		Location ← Location + Offset
	End If
End Sub
```

Continue by refactoring enumerations into tables

When an enumeration is compared directly to a constant in code no matter if you use an "if" or "switch" statement, try to find the property that cause the condition to be satisfied. It is very likely that there will be more constants from the enumeration that will satisfy the same condition once the enumeration has expanded and your condition has long been forgotten. Make a property for each key in the enumeration so that you can't forget to write the values when expanding the enumeration. In another language without the table feature, you can make a function that give the property from the enumeration.

4.2.3. Table style enumeration syntax

Start with the keyword "Key" above the enumerations and add a number of properties after pipes then you can fill each property's column below with final values of the same type for each key in the enumeration. The values in the table cells are stored in a final fixed size array in the enumeration's namespace. The properties can have any datatype. Even collections and other enumerations.

Grammar:

```
Enum <Name>
    Key ( | <PropertyName> As <DataType> )*
    <KeyName> ( , <AliasName> )* ( | <expr> )*
    ...
End Enum
```

Example:

```
Enum Item
    Key     | Sharp As Bool | Hard As Bool | C As Color  | Action As Event
    Arrow   | True          | True         | Color.Black | Shoot
    Stone   | False         | True         | Color.Gray  | Pick
    Pillow  | False         | False        | Color.Blue  | Sleep
End Enum
```

Both "Item.Arrow.Sharp" and "Item.Table[Item.Arrow].Sharp" should return a true boolean value because an arrow is sharp according to the table in the Item enumeration.

The table is treated as an extension to the enumeration because it is easier to learn an extension to something already known than to learn something completely new that is practically the same thing with another name.

4.3. Modularity

Steamroller is a modular language where the only way to connect modules is using the project's module visibility graph.

4.3.1. Content visibility

The keyword **Private** on it's own line make the following declarations invisible to other modules.
The keyword **Public** on it's own line make the following declarations visible to other modules.
The keyword **ScriptEntry** on it's own line make the following methods declared as entry points for scripted execution. This is used for text editing plug-ins.

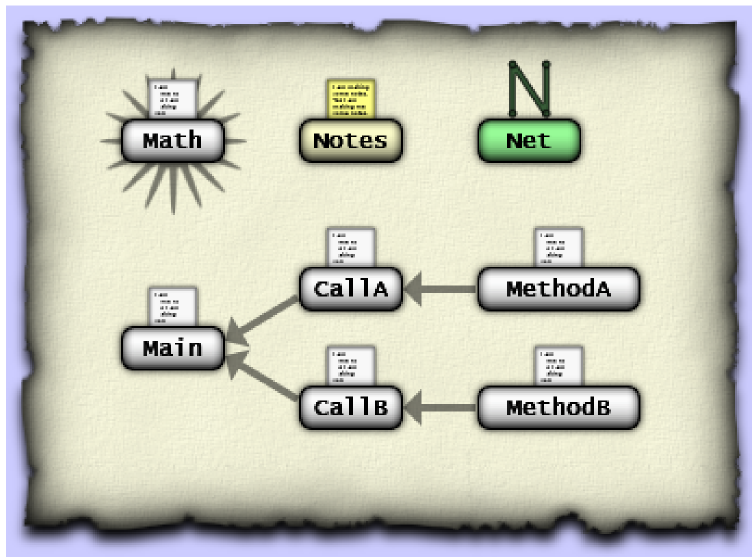
Public is the default setting in the beginning of a new module to make the language possible to use without knowing about private declarations.

Example module:

```
'Everything declared here is public
Private
'Everything declared here is private
Public
'Everything declared here is also public
ScriptEntry
'All methods here can be called as scripts
```

4.3.2. Module visibility graphs

Each project have a graph that connect documents to each other.
An arrow from module A to B means that module B can access things that are public in A. A yellow "sun" module is visible to all other modules in the project (excluding DefLib). This type of module visibility is recommended for math libraries and operative system wrappers that many modules use.



4.3.2.1 A module visibility graph in a project overview

In this example, the module called "Math" have sun visibility and have it's public content visible to all other modules. "Main" can see "Math", "CallA" and "CallB" so that a naming conflict between "MethodA" and "MethodB" can be handled. It is preferred to solve naming conflicts by renaming but this shows how the namespaces works between modules.

4.4. Methods

Steamroller is both procedural and functional using purity levels like in the D programming language. To fully support table-driven programming, method pointers are implemented with full compile time safety. Each method pointer have a template method that define what type of method can be assigned to the pointer. If no method is assigned to the pointer then the template method is used as the default value for doing a default behaviour or showing an error message.

4.4.1. Method inputs

Static arrays can't be passed as input to methods but a type can contain a static array. This is to avoid having to write the same constant range again in the input declaration since the static array's size is not stored in memory. Avoid having too large arrays in types because stack memory have different limits on different systems. Even better is to use dynamic arrays as input to avoid the type wrapping and make the method more reusable. Dynamic arrays can only be passed by reference because otherwise, you would have to make a copy of the whole array to make it safe. Just pass it as ReadOnlyRef if you don't want to modify it. Be aware that you might be modifying it under another name at the same time if you have an aliasing problem where the same array is given as multiple references.

```
Math Fun Foo(Val A As Range, Ref B As Int, Read C As Range) As Int
    Result <- (A.Last + A.First) + B + (C.First + C.Last)
End Fun
```

In this example, the function called "Foo" have math purity so that it can't read from or write to global variables, returns an integer so that the variable called "Result" is an integer and use one of each input access type (Val, Ref, Read).

4.4.2. Input arguments

Access types:

Val (Value)
The input is given as a value so that you can write to it as a local variable without affecting anything outside.
Not recommended for types containing large fixed size arrays because the computer's stack space might be very limited.
Dynamic types and collections will be cloned if needed to prevent aliasing.

Ref (Reference)
The input is an address to a variable and can therefore not allow value expressions as input. Writing to the input will write to the address that the caller gave.

Read (Read only reference)
A reference without permission to write to it.
This makes it possible for the caller to automatically convert values to references by buffering the value and referring to it.
For example, giving 5 + 7 as Read input to a method will compute the value 12 and then refer to 12 with write protection.
The same input can then be used to refer directly to a variable holding the value without making a copy for the most cases.

Multiple input declarations are separated using a comma ",".

Grammar:

<AccessType> <Name> As <DataType>

Examples:

Ref B As Ball, Val Force As Float
Read MyList As Float List
Val A As Int, Val B As Int

4.4.3. Method purity types

Limits how a method can access external variables. Using a high purity level on your methods allow catching many common mistakes in compile time. This will make it much easier to move methods between modules without a global state. Try not to give more data than needed by reference to the methods or you will only move the global state to an object.

Example: A method that apply a modification to a car's wheel would only take a reference to the wheel itself unless the relation to the car is relevant.

Try to group together input arguments with a reusable type if they are too many and related to each other. Vectors of 3 scalars is a common thing that can take advantage of complex types to reduce the risk of entering the arguments in the wrong order. Try to use datatypes declared in standard modules when possible to avoid incompatibility with other people's code. External variables are variables accessed by their global name. If you give the external variable by reference then it counts as accessing an internal variable when you use the internal name.

Options:

Dynamic
Allow both reading from and writing to external variables.
Use this for non reusable code that just need flexibility.
This is the implicit purity that is used when no purity is given because it has no restrictions.

Mute
Allow reading from but not writing to external variables.
Use mute for functions called from boolean conditions when math purity can not be used.

Math
Don't even allow reading from external variables. Use this when you don't want to access anything external by mistake or need it to be called from other methods of math purity.
Use math purity when possible if you write code for others to use since you don't know what purity level they program in.
This is the recommended thing to call from math expressions.
All text input and output have this purity level to allow printing debug information from anywhere.

4.4.4. Binding functions to operands

To keep a minimal syntax, some built-in math operands can be reused for userdefined types by connecting a userdefined function to each combination of input types. Then you just use the math operand like with the built-in datatypes to call your own functions given by the input types.

Unary:
The function must take one input of a type that is used for the matching.

Syntax:

Unary <Operand> alias <FunctionName>

Examples:

Unary - alias MyFunctionName
Unary / alias MyFunctionName
Unary Sgn alias MyFunctionName
Infix Abs alias MyFunctionName
Infix ToString alias MyFunctionName
Infix Normalize alias MyFunctionName
Infix Transpose alias MyFunctionName
Infix Determinant alias MyFunctionName
Unary IsNaN alias MyFunctionName
Unary IsNegInf alias MyFunctionName
Unary IsPosInf alias MyFunctionName
Unary IsValid alias MyFunctionName

Operands beginning with "Is" must be bound to functions returning Bool.
This include IsNaN, IsNegInf, IsPosInf and IsValid.

Infix:
The function must take two inputs of types that are used for the matching.

Syntax:

Infix <Operand> alias <FunctionName>

Examples:

Infix + alias MyFunctionName
Infix - alias MyFunctionName
Infix * alias MyFunctionName
Infix / alias MyFunctionName
Infix ^ alias MyFunctionName
Infix = alias MyFunctionName
Infix <> alias MyFunctionName
Infix < alias MyFunctionName
Infix > alias MyFunctionName
Infix <= alias MyFunctionName
Infix >= alias MyFunctionName
Infix Min alias MyFunctionName
Infix Max alias MyFunctionName
Infix Cross alias MyFunctionName
Infix Dot alias MyFunctionName

4.4.5. Subroutines

Grammar:

```
<PurityType> Sub <Name> ( <Inputs> )
	<Code>
End Sub
```

PurityType is optional and "Dynamic" is the implicit purity.
Exit can be used to jump to the end of the subroutine.

Example:

```
Math Sub FillDynamic(Ref A As Int[])
	Var i as int
	i <- LBound(A)
	loop while i <= UBound(A)
		A[i] <- i * 2 + 1
		i <- i + 1
	end loop
End Sub
```

4.4.6. Functions

Grammar:

```
<PurityType> Fun <Name> ( <Inputs> ) As <DataType>
	<Code>
End Fun
```

PurityType is optional and "Dynamic" is the implicit purity. A local variable called Result is allocated in each function with the same type as the function returns. This variable can be used for reading and writing at any time. Exit can be used to jump to the end of the function and use whatever was previously assigned to Result. When calling a function, the call must be placed within an expression because functions have return values. It is however possible to throw away the result by making a statement that start with "Throw" and continue with the expression.

Example:

```
'Any function named Main will be the main entry point
Dynamic Fun Main() As Int
	'Call MyFunction to execute it's side effects and throw away the result
	Throw MyFunction(X, 5 + 2)

	'Evaluate to 30 and throw away the result in a completely pointless way
	Throw (2 + 8) * 3

	Result ← 0 'The result will be returned to the operating system
End Fun
```

4.4.7. Method pointers

A method is a function or a routine and can be used as a datatype for safe method pointers. Wherever you would place the identifier of the method to call, write Call() around the expression that gives the method pointer dynamically. A function pointer is treated as a value that must always refer to a method that can be called instead of the method that is used as the type. The method that is used as the type will be the default value before it has been given a new method pointer. All type safety is done in compile time so that you don't have to worry about any runtime exceptions other than the ones you make yourself with messages from the template method.

4.4.8. Function pointers

Syntax for calling:

Call (<FunctionPointer>) (<InputArgs>)

Example:

```
Math Fun Op(Val A As Int, Val B As Int) As Int
    PrintString("Called the abstract function.#")
End Fun

Math Fun Op_Add(Val A As Int, Val B As Int) As Int
    Result ← A + B
End Fun

Math Fun Op_Sub(Val A As Int, Val B As Int) As Int
    Result ← A - B
End Fun

Math Fun Eval(Val F As Op, Val A As Int, Val B As Int) As Int
    Result ← Call(F)(A, B)
End Fun
```

4.4.9. Routine pointers

Syntax for calling:

Call (<RoutinePointer>) (<InputArgs>)

Example:

```
Var X as Int
Fixed Sub SetInt(Val NewValue as Int)
    PrintString("Called the abstract method ""SetInt"".#")
End Sub
Fixed Sub SetInt_X(Val NewX as Int)
    X ← NewX
End Sub
Dynamic Fun Main() as Int
    'Assign SetX as the set method
    SetMethod ← SetInt_X

    'Call the set method using 5 as input
    Call(SetMethod)(2 + 3)

    'X is now 5
End Sub
```

4.5. Math

An important task of a programming language is to define a standard for communication between modules developed by different teams. Steamroller has optional standard math libraries for linear algebra and fractals that are trying to prevent projects from reinventing the basic types. Binding of functions to operands makes sure that they feel like a built-in part of the language and allow userdefined math libraries to do the same. The most unconventional math operand is the unary reciprocal that allow writing "/X" instead of "1.0 / X" just like you would write "-X" instead of "0.0 - X" to save a lot of space. Just remember not to write like that on a math exam.

4.5.1. Order of computation

1. From left to right

() Function calls with arguments within parentheses to the right of the method's name

Example: Foo(5,2)

[] Array lookups with index expression within parentheses to the right of the method's name

Example: MyArray[n] <- MyArray[n+1]

2. From left to right

"." attribute lookups

Example: Vector.Y

3. From right to left

"-" unary negation of numeric values

Example: -5

0 - 5 = -5

"/" reciprocal

Write /X instead of 1/X to get the reciprocal

Example: /2

1 / 2 = 0.5

"Not" unary negation of boolean values

Example: Not X > 4

4. From left to right

"^" raise to the power of

Example: 3 ^ 2

3 ^ 2 = 3 * 3 = 9

5. From left to right

"*" multiplication

Example: x * y

"/" division

Example: x / y

"IntDiv" integer division

Example: x IntDiv y

"Mod" modulo

Example: x Mod y

"Cross"

Cross product

"Dot"

Dot product

6. From left to right

"+" addition

Example: x + y

"-" subtraction

Example: x - y

"&" list concatunation

Concatunate 2 lists of the same fixed element type.

Examples:

LeftList & RightList

"Hello " & YourName & "!"

"Hel" & "lo"

Constant strings are concatunated during tokenization to save time on concatunation. It is allowed to use the line continuation token "...", a comment behind "'" and a linebreak between the constants strings without concatenating in runtime.

7. From left to right

"Min" numeric minimum

Example: 2 Min 6

2 < 6 and therefore 2 is the minimum

Example: 8 Min 8

8 = 8 and therefore 8 is the minimum

"Max" numeric maximum

Example: 5 Max 7

7 > 5 and therefore 7 is the maximum

Example: 4.5 Max 2.7

4.5 > 2.7 and therefore 4.5 is the maximum

8. From right to left

"=" equality returns true iff both sides are equal

Not recommended to use with float because of rounding errors

"<" lesser returns true iff left side is smaller than the right side

">" greater returns true iff left side is more than the right side

"<=" lesser returns true iff left side is smaller than or equal to the right side

">=" greater returns true iff left side is more than or equal to the right side

9. From left to right

"And" logical and returns true iff both sides are true

False And False = False
True And False = False
False And True = False
True And True = True

"Or" logical and returns true iff any sides are true

False Or False = False
True Or False = True
False Or True = True
True Or True = True

"Xor" logical and returns true iff exactly one side is true

False Xor False = False
True Xor False = True
False Xor True = True
True Xor True = False

".." range constructor

An infix constructor for the range datatype.

10. From left to right

"," separator for arguments

Examples:

```
Math Fun Foo(Read A As Int, Read B as Int) As Int
Result <- Foo(2, X)
```

11. Statement

This step is no longer in expressions but they still need to be documented.

"<-" assignment from right value to left reference.

Both sides must have the same type. The expression to the left must be possible to write to and the expression to the right can be any expression of a compatible datatype. You can not assign 2 + 2 to 4 for example because 4 is a value. If the target is a write protected reference then assigning to it is not allowed.

"<->" swap memory between 2 references.

Both sides must have the same type. This do not require any garbage collection because swapping does not delete or duplicate anything. You can apply it to any variables given as writable references even if they are given from collections or are whole collections by themself.

4.5.2. Expressions

It is recommended to use parentheses to avoid misunderstandings even if the compiler know the precedence of the operations.
For example, it is better to write "X + (C * 4)" than "X + C * 4" because "X + (C" can't be mistaken for "(X + C)" when doing fast replacement of sub expressions.

When the order does not affect the result, parentheses will only be in your way.
"A + B + C + D" is better to write than "(((A + B) + C) + D)".
The same applies to "*", "and", "or", "xor", "min" and "max" that are also independent of order unless you add floats of very different size or assigned your own functions to the operands.

Named unary math operands require that you use parentheses around the input.
This is to avoid being white space sensitive in cases like "Sin(X)" where "SinX" would be tokenized like one identifier after removing the white space by mistake.

Examples:

```
True 'Constant boolean value
False 'Constant boolean value
64 'Constant integer
1.76 'Constant floating point value
-X 'Negation
/X 'Reciprocal
A + B 'Addition
A - B 'Subtraction
A * B 'Multiplication
A / B 'Division
A ^ B 'A raised to the power of B
A Mod B 'A in modulo B
A IntDiv B 'Truncated integer division
A Min B 'Minimum of A and B
A Max B 'Maximum of A and B
Sqr(X) 'Square root of X
Sin(X) 'Sinus of X
Cos(X) 'Cosinus of X
Tan(X) 'Tangent of X
ArcSin(X) 'Inverse sinus of X
ArcCos(X) 'Inverse cosinus of X
ArcTan(X) 'Inverse tangent of X
HypSin(X) 'Hyperbolic sinus of X
HypCos(X) 'Hyperbolic cosinus of X
HypTan(X) 'Hyperbolic tangent of X
Ceil(X) 'X rounded up
Floor(X) 'X rounded down
Log(X) 'Natural logarithm of X
Log10(X) 'Base 10 logarithm of X
Abs(X) 'Absolute of X
Sgn(X) 'Signum of X
```

4.5.3. Extreme values

0.0 / 0.0 returns NaN_Float that will return false in all comparisons except IsNaN. Check if a Float value is NaN (not a number) using the IsNaN function.

1.0 / 0.0 returns PosInf_Float that is more than all other values.
-1.0 / 0.0 returns NegInf_Float that is less than all other values.
IsNaN(0.0 / 0.0) returns true because the input is not a value.
IsNegInf(-1.0 / 0.0) returns true because the input is a negative infinity.
IsPosInf(1.0 / 0.0) returns true because the input is a positive infinity.
IsValid(26.0) returns true because the input returns false from IsNaN, IsNegInf and IsPosInf. Use IsValid to check if the output from a calculation can be used as a real number.

Sometimes infinity is needed as a default value when getting the minimum or maximum of a collection. Then infinity is a good error value to say that the function did not get a set any data. Even NaN_Float can be useful if the value is unknown but allowing NaN should be well documented in the postcondition.

4.5.4. Comparisons and boolean operations

Remember to use a treshold when comparing floating point values.

X = Y should be written as Abs(X - Y) < Treshold.

Examples:

```
A = B 'Equal to comparison
A <> B 'Not equal to comparison
A > B 'Greater than comparison
A < B 'Lesser than comparison
A >= B 'Greater than or equal to comparison
A <= B 'Lesser than or equal to comparison
A = B and B = C 'Boolean and returns true iff both returns true
A = B or B = C 'Boolean or returns true iff one or more returns true
Not A 'Boolean not returns true iff A is not true
```

4.6. Line breaks

The language is depending on line breaks to divide statements into logical lines. The continuation token "..." must be placed before the linebreak to continue a logical line on the next physical line. If the continuation token is written after a "." or ".." token then you must separate them with white space. It is okay to place comments using "'" after the continuation token.

Example:

```
X ← Foo(A * 2, B - 3, ...   'You can place a comment here
C + 1, D * 8, E, F * F)     'You can place a comment here
```

The parser will only see "X <- Foo(A * 2, B - 3, C + 1, D * 8, E, F * F)" from the tokenizer as if the linebreak never existed.

Dividing statements by linebreaks like in Visual Basic remove the need for ending each statement with ";". Personally I think that it makes the code more clean and readable since ending a statement should be the same as ending a line as long as you don't run out of horizontal space.

4.7.1. Collection access

Fixed size arrays, dynamic arrays and lists can be accessed using an integer index expression within square brackets.

Examples:

MyFixedSizeArray[8]
MyDynamicArray[A[B[N] - B[N – 1]]]
MyList[(X / 2) + Y]

Dynamic array keyword methods

Resize MyDynamicArray to MyRange
Reallocates the buffer in MyDynamicArray to MyRange.

List keyword methods

Resize MyList to MySize
Reallocates the buffer in MyList to 1..MySize.
Any elements that are outside of the new range will be deleted.

Clear MyList
Sets the list's length to 0 without removing the memory allocation.

Push MyElement to MyList
Inserts MyElement to a new slot in the end of MyList.
The end is the default because it is both fastest and preserve
the order of the previous elements.

ShiftInsert/**SwapInsert** MyElement To MyList At [Index]
Inserts MyElement to a new slot
Methods:
ShiftInsert is used to preserve the order of the old elements.
Like sneaking into a line of people.
SwapInsert is used to replace whatever existed in that slot
and place the old element at the end. Like pushing out
someone from a line of waiting people.

ShiftRemove/**SwapRemove** [Index] From MyList
Delete the selected element from MyList.
Methods:
ShiftRemove is used to preserve the order of the remaining elements.
Like walking out of a line of people.
SwapRemove is used to place the last element in the hole to save time
but destroy the order. Like giving your waiting ticket to the person that
just came in because you don't need it.
Referencing:
[MyIndex] removes the element at MyIndex.
Using a reference to the data in the element is not supported
because it would be dangerous.

4.7.2. Collection declaration grammar

Where MyName is the variable's name, MyElementType is the element's type and MyConstantRange is the fixed array's range.

Lists:

Var MyName as MyElementType List

The list is the most easy collection to use because it allocate more memory automatically when inserting new elements.

Examples:

Var Balls as Ball List

Var IndexList as Int List

Resizing uses an integer to specify the number of allocated slots for element. If the new buffer length is too small to contain the existing elements then the buffer will use the smallest size that can hold the existing data.

Setting the buffer length to 0 will free the dynamic allocation if there are no elements and remove all padding if there are elements.

Examples:

Resize Balls to 64

Resize MyList to N * M

Dynamic arrays:

Var MyName as MyElementType[]

The dynamic array allows specifying exactly when and how to reallocate to get speed at the risk of making more mistakes. Usually, you have a variable telling what parts of the array is being used and reallocate when the space is running out.

Examples:

Var Buffer as DataSample[]

Resizing:

Resize MyDynamicArray to -5..5

Resize Buffer to 1..(N * 128)

Fixed size arrays:

Var MyName as MyElementType[MyConstantRange]

Fixed size arrays are good when the collection naturally have a fixed upper bound so that you don't waste any space on padding or time on allocating. Fixed size arrays with fixed size elements can be declared in fixed types.

Examples:

Var Wheels as Wheel[1..4]

Var AccidentsPerWeekday as Int[1..7]

4.8. Identifiers

Any name that you give yourself is an identifier that must follow these rules.

The name:

* Must have at least one character.
* May not contain double underscore "__" because double underscore is used to separate namespaces.
* May not begin or end with underscore "_" because that would be combined with the double underscore used for namespaces.
* May only contain digits (0..9), latin characters (a..z and A..Z) and underscore "_" to make it easy to compile to C code.
* May not start with a digit because that would create an ambiguity in the generated C code.

4.9. Strings

Write "YourQuotedText" to crate a constant list of integers representing the quoted text. Press Ctrl + Return to write a quoted line break in your constant string. The character is stored as code 31 in the document and translated into code 10 in the final string. This is not compatible with regular text editors. Write 2 quote signs in a pair to write a quoted quote sign that will not end the current quote.

Example:

```
PrintString("My name is """ & MyName & """.")
```

Call **PrintString** with a string expression as input to print the string to any available text console or show it as a message box if running as a script.
Try to avoid printing special characters to ensure consistent behaviour on different platforms.

Call **GetString** as a function to ask the user for a string.
Since strings are lists of integers, it can concatenate strings like any other list.

Example:

```
"123" & "4567" is the same as "1234567"
```

4.10. Types

Declared variables must have types. The "Range" type is already built-in to the core of the language and is created using the infix .. operator. Types can contain variables of other types but fixed types can not contain dynamic types or dynamic collections. Putting a fixed size array in a type allow it to be returned by value from functions. Functions can not return fixed size arrays that are not contained in a type because of limitations in C99. Dynamic collections are okay to return from functions because the actual value returned is just a tiny collection head pointing to a dynamic allocation.

Examples:

```
Fixed Type Area
	Var X as Range <- 0..-1
	Var Y as Range <- 0..-1
End Type
Fixed Type Volume
	Var X as Range
	Var Y as Range
	Var Z as Range
End Type
Fixed Type MyFixedArray
	Var A as Int[-5..5]
End Type
Type MyDynamicArray
	Var A as Int[]
End Type
Var A as Range <- 1..5
Var B as Area <- Area(1..10, 1..10)
Var C as Volume <- Area(-5..5, -5..5, -5..5)
```

4.10.1. Type conversion

Do not use floating point values for holding integers since the precision depends on the distance from zero and rounding may cause even more loss of precision. ToString is an operation that can be used instead of IntToString, FloatToString, BoolToString, CharToString or any userdefined conversion to a string that is bound to ToString as an alias.

Examples:

```
FloatToInt(3.26) 'Rounded to the closest integer 3
FloatToInt(2.999) 'Rounded to the closest integer 3
FloatToInt(1.5) 'Rounded to the closest integer 2
FloatToInt(-1.999) 'Rounded to the closest integer -2
IntToFloat(54) 'Converting to the floating point value 54.0
IntToFloat(-270) 'Converting to the floating point value -270.0
IntToString(-64) 'Generating the string "-64" as a "Char List"
StringToInt("378") 'Returning the integer 378
```

4.11.1. Assignment

Write "<-" between your assignment target and the new value to assign. The editor will recognize it and draw it as a real arrow. You can copy the content of a list to another list by assigning one list to the other. This does not work with arrays because arrays don't know which elements are actually used.

Grammar:

```
<RefExpr> <- <Expr>
```

Examples:

```
i <- i + 1
ListA <- ListB
WhoAreYou <- "Who" & " are" & " you?"
    Constant strings are write protected integer lists that can be
    concatunated using the list concatunation operand "&".
MyArray[x + 5] <- (17 + Y) * 4
BallA <- BallB
Ball.Diameter <- Ball.Radius * 2
Table[Index[i]].Children[j] <- Content[i] * Content[i + 1]
```

4.11.2. Swap assignment

Write "<->" between 2 reference expressions to swap memory between. The editor will recognize it and draw it as a real double-sided arrow. Any datatypes are accepted as long as both sides have the same type. Swap assignments will never cause the need for any cloning or garbage collection because nothing was duplicated or destroyed. This makes a swap much faster than an assignment for dynamic data structures by not even touching their dynamic allocations.

Grammar:

```
<RefExpr> <-> <RefExpr>
```

Examples:

```
Var Left as Int
Var Right as Int
Left <- 1
Right <- Length(Result)
Loop until Left >= Right
    Result[Left] <-> Result[Right]
    Left <- Left + 1
    Right <- Right - 1
End Loop
```

4.12. Control flow

The thing that makes imperative programming so powerful is the ability to modify the program counter and do the work of a state machine with jumps between blocks of machine code.

4.12.1 If statements

The most basic type of structured control flow that works exactly as in Basic.

Example:

```
If X - 1 > 7 + 2 Then
	'Do something when X - 1 is greater than 7 + 2
End If
```

Example:

```
If A = B Then
	'Do something when A equals B
Else
	'Do something when A does not equal B
End If
```

Example:

```
If A = B Then
	'Do something when A equals B
ElseIf C = D Then
	'Do something when C equals D but A does not equal B
Else
	'Do something when all previous cases are false
	'Recommended to use if you want to see all execution paths
End If
```

4.12.2. Goto

Gotos can be used to jump to another place when you can't find a higher statement for it. It is in general bad practice to use the goto statement but error handling is often messy no matter how you implement it.

Example:

```
Dynamic Fun Main() As Int
	Result ← 10 'Set the result to 10
	Goto X 'Jump from here...
		Result ← Result + 1
	Place X '...to here
	Result ← Result + 1 'Add 1 to the result
	'Result should now be 11
End Fun
```

4.12.3. The while loop

A more complete version of the language could allow having the condition at the end of the loop to always execute once before evaluating the condition but I did not have the time to implement all the standard features from other languages. Switch statements are also missing in the prototype because implemeting that feature would be an exam project by itself since it is mostly a performance optimization.

<Code> is executed in a loop while <Boolean> is true.

<Boolean> is evaluated before each iteration of the loop.
Changing a variable to make <Boolean> false will continue the current iteration before evaluating <Boolean> again.

Syntax:

```
Loop While <Boolean>
    <Code>
End Loop
```

Example:

```
Loop While True
    'This loop will never end by itself
    'This is common when running a game loop.
End Loop
```

Example:

```
Var I as Int
I <- 1
Loop While I <= 10
    'I is moving from 1 to 10.
    'I start at 1 and is increased while (I <= 10) is true.
    'Any function calls in the boolean expression after the While
    'keyword will be executed in each iteration of the loop.
    I <- I + 1
End Loop
```

4.12.4. The until loop

Just like the while loop but with a negation applied to <Boolean>.

Syntax:

```
Loop Until <Boolean>
	<Code>
End Loop
```

Example:

```
Var I as Int
I <- A
Loop Until I > B
	'I is moving from A to B.
	'I start at A and is increased while (I > B) is false.
	I <- I + 1
End Loop
```

4.12.5. The for loop

<Var> is the variable to iterate over <Range> while <Code> is executed in a loop. You may not use more than one token to identify the variable because otherwise the compiler would have to execute the expression multiple times with duplicated side effects and slow speed or store a pointer to the variable that might be outdated after reallocating it's memory owner.

<Range> evaluated before the first iteration and stored to a hidden variable. Feel free to generate the range expression from time consuming function calls since it is only evaluated when the loop start.

<Step> is the constant integer to add after each iteration.
A negative step will generate a backward loop and <Step> must therefore be a constant integer expression.

Do **not** use a backwards <Range> for anything else than representing an empty range. Even when when using a negative <Step>. The backward for loop start at the last index and end at first index. This is because the empty range is (0..-1) and (-1..0) would iterate over the 2 values {-1, 0} instead of nothing.

Use the while or until loop if you need more power than a for loop can give.

Syntax without step:

```
For <Var> <- <Range>
    <Code>
End For
```

Syntax with step:

```
For <Var> <- <Range> Step <Step>
    <Code>
End For
```

Example:

```
Const StartIndex as Int ← 2
Const EndIndex as Int ← 8
Final IndexRange as Range ← StartIndex..EndIndex
Dynamic Fun Main() As Int
    Var I as Int

    For I ← StartIndex..EndIndex
        'I is moving from 2 to 8.
    End For

    For I ← IndexRange
        'I is moving from 2 to 8.
    End For

    For I ← 1..10 step 2
        'I is moving from 1 to 10 but skip every even number.
    End For

    For I ← 1..10 step -1
        'I is moving backwards from 10 to 1.
    End For
End Fun
```

4.13. Using applications as functions

A very old feature from the early operating systems is the ability to give input arguments as a list of strings and return an integer. The input is usually for flags and filenames for loading something. The output is usually the error code with 0 for success and other values for error codes.
Steamroller allow inserting a line of C code after the "#" sign to get full access to the file system with all the low level datatypes and pointers needed for effective file handling.
Any file handling built-in to Steamroller would be too limited for existing file formats because of the high level datatypes. For programmers that don't know C, different wrappers can be made for handling text files and streaming of raw data.

4.13.1. Application input

When an application is started by the operating system, a list of arguments can be given as input. This is especially useful if you started the appilcation from a command line interpreter or another application.

There is a global list of command line arguments that you can read from.
Hidden declaration:

```
Final CommandLineArguments As String List
```

This is to hide away complexity for most of the applications that don't need command line arguments.

The "String" type only contain a char list because the language requires all collection element types to be named.
Hidden declaration:

```
Dynamic Type String
    Var Chars As Char List
End Type
```

4.13.2. Application output

The "Main" function must return an integer to the operative system. This is often used for error messages but you can use it for anything you want.

5. Modifying programming on the C level

If you insert your own C code or modify the compiler then:

Try to use good prefixes for global names to avoid name collisions.

Tripple underscore "___" is used after a prefix to reserve something for the language.

Double underscore "__" is used after a module name.

Single underscore "_" may be used by the programmer.

Call "M___Alloc" instead of "malloc" and "M___Free" instead of "free".

This will help you to detect memory leaks if you forget to free and the methods can easily be replaced with a custom allocator if you want more speed at the expense of reserving more memory.

If the application leaked a positive number of allocations then you probably:

* Forgot to free an allocation.

If the application leaked a negative number of allocations then you probably:

* Tried to free an unused pointer.
* Duplicated a pointer without cloning the allocation it pointed to.

If you create your wrappers for existing C libraries then:

* Avoid or wrap libraries that require pointers since they will give you random crashes and memory leaks like when programming in C.
* Prefer safe and well tested libraries that allow fast development.
* Make the method declaration in Steamroller but call the C library direcly using C. Everything on a line after the "#" sign will be pasted directly into the resulting C code. Methods that include C code can not be called when running the code in the virtual machine.
* Remember the "L___" prefix that is given to local variables. Using local variables is preferred to allow renaming the module and to avoid flooding the global namespace in C.
* Remember that methods also get a prefix that depend on what the current module is named.
* Try to have one layer of pure wrapper methods and one layer of high level helping methods. Don't try to add C code in type declarations. Don't use C code for jumping to labels or exiting the method directly. To exit the method, use the high language to exit so that the Exit label is created.

6. Conclusion and discussion

How can a programming language define the table driven paradigm?
The table feature in Steamroller is a good start by improving safety and readability but a single feature is not enough to define the table driven programming paradigm. A paradigm defining language usually have 2 or 3 features to support the paradigm. A paradigm defining version of the Steamroller programming language would use the same dot notation for accessing dynamic lists as tables and allow storing the content in a database system that handle version conflicts.

How can professional programming move closer to graphical programming without being too limited?
By mixing with written code and always show a preview of generated code in the written programming language for users that don't understand how graphs are integrated with written code.

Following the same principle in interface design would let the programmer interpret the data given in the interface designer. Interface design tools like in Microsoft's Visual Studio usually take away too much power to be useful in advanced interfaces. The designed interfaces are not possible to mix with your own graphics engine and put too many limitations on the component model.

7. Future development

A more complete version of the language would reduce the refactoring cost from fixed size table to a **dynamic table** stored in a file by allowing the table to be replaced with something dynamic without changing the syntax for accessing the content. The simplest way might be to connect an index variable to a list to access the list's content from the index. The problem is that it would have to be combined with object orientation to identify the instance of the table when the table is not constant.

The system for **guessing what the programmer is trying to write** can be improved a lot using machine learning that remember past mistakes. The current version is only a rough approximation written in a short amount of time.

Extending the language with table driven programming on the **graphics card** would improve performance a lot. Shader model 5 hardware would be the minimum requirement to avoid the time and output size limits in shader model 4.

Calling a data flow graph from another data flow graph is a bit messy right now because the generated code use side effects for realtime behaviours but the nodes require functions. The data flow graphs could be extended to allow inserting another data flow graph as an **integrated circuit** with inputs and outputs on the top and bottom side. The data in the integrated circuit would be allocated in the node. This modularity is common in electronic simulators and a must for larger systems.

Book references

About Assembler
Kip R. Irvine (Mar 7, 2010). Assembly Language for x86 Processors. ISBN-13: 978-0136022121

About Algol
Daniel D. McCracken (Dec 1962). A Guide to Algol Programming. ISBN-13: 978-0471582342

About Lisp
Conrad Barski (Nov 15, 2010). Land of Lisp: Learn to Program in Lisp, One Game at a Time. ISBN-13: 978-1593272814

About Simula
Bjørn Kirkerud (Nov 1989). Object-Oriented Programming With Simula. ISBN-13: 978-0201175745

About Pascal
Larry R. Nyhoff and Sanford Leestma (Mar 1988). Advanced Programming in PASCAL with Data Structures. ISBN-13: 978-0023695506

About Prolog
Ivan Bratko (Sep 8, 2000). Prolog Programming for Artificial Intelligence. ISBN-13: 978-0201403756

About SQL
Ben Forta (Nov 4, 2012). Sams Teach Yourself SQL in 10 Minutes. ISBN-13: 978-0672336072

About Renderman
Don Rudy Cortes and Saty Raghavachary (Dec 31, 2007). The RenderMan Shading Language Guide. ISBN-13: 978-1598632866

A book about evaluating usability with knowledge instead of users
Larry L. Constantine and Lucy A. D. Lockwood (Apr 17, 1999). Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design. ISBN-13: 978-0201924787

A book that covers the common design principles and evaluation methods for interfaces
David Benyon (Mar 16, 2010). Designing Interactive Systems: A Comprehensive Guide to HCI and Interaction Design. ISBN-13: 978-0321435330

Web references

Why and when to use table driven programming.
Dave Thomas (2005). Agile Programming: Design to Accommodate Change.
martinfowler.com/ieeeSoftware/accChange.pdf

A computer magazine from the early days of programming that
shows what programmers had problems with at the time.
Computer World (Jul 26, 1967).
http://news.google.com/newspapers?
nid=v_xunPV0uK0C&dat=19670726&printsec=frontpage&hl=en

A good online book about shader programming and data driven computations.
NVIDIA corporation (2007). GPU gems 3.
http.developer.nvidia.com/GPUGems3/gpugems3_ch01.html