

Введение

Проект *GW-Basic* представляет собой среду разработки приложений, а также диалект языка программирования *Basic*. Среда разработки предоставляет пользователю возможности для написания программ на одноименном языке, вычисление выражений путем интерпретации запроса после ввода, а также визуализацию графики.

В данной работе рассматривается реализация среды исполнения *GW-Basic* с использованием средств генерации анализаторов одноименного языка (инструменты *flex*, *bison*), а также создание графической оболочки для взаимодействия со средой средствами *OpenGL* и инструментарием *glut*.

1 Обзор используемых технологий

1.1 Среда разработки GW-Basic

Среда разработки Microsoft GW-Basic, а также одноименный язык программирования представляют собой платформу для написания программ в императивной нотации. Управление средой осуществляется с помощью командной строки, в которую попадает пользователь после запуска.

1.1.1 Синтаксические конструкции

Программа на GW-Basic может включать в себя следующие синтаксические конструкции [3]:

- **Ключевые слова** (англ. *Keywords*) – представляют собой зарезервированные слова среды исполнения GW-Basic, и являются частью операторов или команд.
- **Команды** (англ. *Commands*) – это исполняемые инструкции. Выполнение команд осуществляется сразу после ввода.
- **Операторы** (англ. *Statements*) – являются исполняемыми инструкциями программы на GW-Basic. Представляют собой группу ключевых слов, используемых как строки программы среды GW-Basic.
- **Функции** (англ. *Funtions*) – по типу возвращаемых значений могут быть: строковыми, численными.
- **Переменные** (англ. *Variables*) – определенная строка, за которой установлено определенное значение. Переменные могут быть объявлены/изменены как пользователем, так и контекстом программы.

Ключевые слова (имена команд, операторов) представляют собой последовательность заглавных латинских букв. Примерами ключевых слов в GW-Basic являются слова: PRINT, RETURN, GOTO. Ключевые слова не могут быть использованы в качестве имен переменных, иначе это бы привело к конфликту с такими синтаксическими конструкциями, как *Команды* и *Операторы*.

Имена функций и переменных могут включать в себя: буквы латинского алфавита, цифры, символ ' '. Также, имена могут начинаться только с латинской буквы.

Список всех синтаксических конструкций представлен в [3, стр. 117].

1.1.2 Режимы интерпретации запросов

Интерпретация пользовательских запросов в среде GW-Basic может проходить в следующих режимах:

1. **Прямой** (англ. *Direct*)
2. **Непрямой** (англ. *Indirect*)

В *прямом режиме*, введенные операторы и команды исполняются сразу после окончания ввода. Этот режим используется преимущественно в целях отладки программы, либо для вычисления выражений, для которых нет необходимости писать программу.

Непрямой режим используется для создания/редактирования строк программы. Каждая строка программы на GW-Basic имеет следующий формат:

`nnnnn statement[statements]`

Где `nnnnn` – номер строки, а `statement` – оператор GW-Basic. В зависимости от логики программы, строка может содержать более одного оператора (`[statements]`). В этом случае, операторы должны быть разделены символом двоеточия ':'. Для запуска программы, используется команда `RUN`.

Полное руководство по редактированию программы в среде GW-Basic представлено в [3, стр. 18].

1.1.3 Графический режим

Согласно [3, стр. 142], среда выполнения GW-Basic имеет несколько режимов визуализации информации на экране. Использование команды `SCREEN` позволяет изменять режим вывода среды с целью включения/отключения/изменения графического режима. По умолчанию, т.е. при запуске среды без использования этой команды, среда работает в *текстовом режиме*, что означает, что выполнение любого оператора, связанного с графикой, будет проигнорировано.

При активации *графического режима*, пользователю становится доступно использование графических операторов. Среда GW-Basic предлагает визуализацию следующих примитивов: CIRCLE, LINE, POINT, и т.д. Полный список операторов представлен в [3, стр. 117].

1.2 Flex и Bison – инструменты разработки анализаторов

В процессе развития теории построения компиляторов, сформировалось множество подходов к обработке и анализу программ различных языков программирования. Наиболее популярный из них заключается в разбиении задачи разбора на два этапа [4, стр. 21]:

1. **Лексический анализ программы** (англ. *Scanning*) – сканирование текста программы с целью выделения *токенов* (англ. *Tokens*), а также значений, которые стоят за этими токенами.
2. **Синтаксический разбор** (англ. *Parsing*) – установление связей между токенами, называемых *правилами грамматики*.

Для выполнения лексического разбора, наиболее популярным генератором анализатора является *flex*. В основе выделения токенов лежит использование регулярных выражений. Что касается генерации синтаксического анализатора, то для этих целей широко применяется *bison*. Для описания связей между токенами, *bison* использует нотацию правил грамматики в форме Бэкус-Наура.

1.3 Glut – инструментальный для работы с OpenGL

Glut представляет собой оконно-независимый инструмент для написания программ на *OpenGL*. Инструментарий поддерживает *функции обратного вызова* (англ. *Callback functions*) для большинства событий [1]. В частности, наиболее используемые функции обратного вызова:

- **glutDisplayFunc** – вызывается при отрисовке сцены.
- **glutReshapeFunc** – используется для обработки события изменения размера окна.
- **glutKeyboardFunc** – вызывается в случае возникновения нажатия клавиши.

2 Разработка среды GW-Basic

2.1 Анализаторы языка GW-Basic

2.1.1 Лексический анализатор

Рассмотрим основные конструкции, которые необходимо выделить в лексемы:

1. Ключевые слова
2. Имена функции и переменных.
3. Константные значения: численные и строковые.
4. Символы (арифметических операций, операций сравнения, операторы логики).

Создание переменных или функций, названия которых совпадает с названием ключевых слов недопустимо [3]. Это означает, что приоритет за определением ключевых слов, и что разбор имен функций и переменных должен идти после разбора ключевых слов. Что касается константных значений и символов, то распознавать эти синтаксические конструкции можно на любом этапе.

2.1.2 Синтаксический анализатор

Целью использования синтаксического анализатора является построение *дерева синтаксического разбора* (англ. *AST-Tree*). Поскольку среда разработки GW-Basic представляет собой командную строку, то синтаксический разбор должен производиться для каждого пользовательского запроса. Учитывая режимы интерпретации команд (см. п. 1.1.2), сокращенная форма грамматики в нотации Бэкус-Науровой формы, выглядит следующим образом (см. листинг 1):

Листинг 1: Грамматика синтаксического анализатора языка GW-Basic.

```
<Interpreter> ::= <DirectMode> | <IndirectMode> // Корневой нетерминал

<DirectMode> ::= <Command>
<Command> ::= <Run> | <System> | ...
...
// Нетерминалы команд
```

```

...
<IndirectMode> ::= <LineNumber> <Statements>
<Statements> ::= <Statement> ':' <Statements> | <Statement>
<Statement> ::= <Beep> | <Call> | <Circle> | <Line> | ...
...
// Нетерминалы операторов
...

```

Грамматика для переменных языка GW-Basic представлена в листинге 2. Под терминалом **DECLARATION** понимается объявление имени функции или переменной. Символы, следующие за этим терминалом, указывают на тип значения переменной.

Листинг 2: Грамматика для переменных языка GW-Basic.

```

<Variable> ::= <StringVariable> | <NumericVariable>
<StringVariable> ::= DECLARATION '$' // Строковый тип
<NumericVariable> ::= DECLARATION '%' | // Целочисленный тип
                        DECLARATION '!' | // Вещественный тип, одинарная точность
                        DECLARATION '#' // Вещественный тип, двойная точность

```

Для вычисления значений, как с целью вывода результата, так и для определения присваемого значения переменной, необходимо определить грамматику *выражений* (англ. *Expressions*) (сокращенный вариант, см. листинг 3). Поскольку значения переменных могут быть строкового или численного типа, то и выражения могут быть тоже двух типов.

Листинг 3: Грамматика для выражений языка GW-Basic.

```

<Expression> ::= <NumericExpression> | <StringExpression>
<NumericExpression> ::= <ArithmeticalOperator>
                        | <LogicalOperator>
                        | <FunctionalOperator>
                        | <RelationalOperator>

<StringExpression> ::= <StringOperator>
<StringOperator> ::= <StringTerm> '+' <StringOperator>
                    | <StringTerm>
<StringTerm> ::= <StringVariable>
                | STRING_CONSTANT

```

Для строк, среда GW-Basic предоставляет единственную бинарную операцию – конкатенацию. В свою очередь, численные выражения разбиваются на четыре класса (см. [3, стр. 46]):

- **Арифметический оператор** – выполнение бинарных операций '+', '-', '^', '/',.
- **Логический оператор** – выполнение булевских операций NOT, AND, OR, XOR, и т.д.
- **Функциональный оператор** – математические/строковые функции.
- **Оператор отношения** – выполнение бинарных операций '>', '<', '=', '<=', '>=', и т.д.

2.2 Архитектура системы исполнения

Архитектура системы исполнения представлена на рис. 1. Компоненты системы исполнения представлены блоками, а зависимость между блоками обозначена стрелочками.

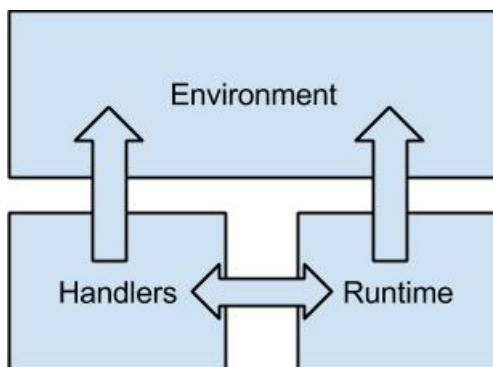


Рис. 1 – Архитектура системы исполнения

Рассмотрим компоненты системы исполнения подробнее.

2.2.1 Окружение среды

Окружение (англ. *Environment*) включает в себя структуру, которая хранит текущее состояние интерпретатора, а также набор функций для изменения содержимого структуры. Чтобы описать текущее состояние, структура должна содержать следующую информацию:

- Режим работы интерпретатора.
- Привязку к системе вывода/вывода.
- Контекст выполнения.

Интерпретатор среды должен поддерживать следующие режимы работы (согласно п. 1.1.2):

1. Интерпретацию команд.
2. Выполнение программы на GW-Basic.

2.2.2 Обработка пользовательских запросов

Компонент *обработки вершин дерева синтаксического разбора* (англ. *AST-Node Handler*) (см. рис. 1, *Handlers*), представляет собой набор функций для обработки вершин дерева. Под процессом обработки вершины понимается внесение изменений в *контекст исполнения* (см. п. 2.2.1). Так, для каждого типа вершины дерева, существует в точности единственная функция обработки.

Каждая функция обработки, должна иметь следующие входные параметры:

- Окружение системы исполнения.
- Вершину дерева синтаксического разбора, тип которой соответствует типу функции.

Поскольку вершина является элементом дерева, то обработка дочерних вершин осуществляется методом рекурсивного спуска.

2.2.3 Выполнение команд и программы

За выполнение пользовательских команд отвечает *система времени выполнения* (англ. *Runtime*) (см. рис. 1). Согласно п. 1.1.2, синтаксис записи команд влияет на режим выполнения этих команд средой. В тоже время, режимы выполнения учитываются грамматикой синтаксического анализатора (см. п. 2.1.2), поэтому реализация *прямого* и *непрямого* режимов полностью ложится на обработчики соответствующих вершин дерева синтаксического разбора.

Таким образом, процесс выполнения пользовательского запроса состоит из следующих этапов:

1. Чтение пользовательского запроса.
2. Применение анализаторов (п. 2.1.1, 2.1.2) с целью построения дерева синтаксического разбора.
3. Обработка корневого узла дерева с помощью соответствующего обработчика.

Рассмотрим особенности выполнения программы на GW-Basic. В процессе выполнения программы, может возникнуть необходимость ее временной остановки. Пример такой ситуации – ожидание ввода значения переменной-аргумента оператора `INPUT`. Таким образом, процесс выполнения программы на GW-Basic состоит из следующих этапов:

1. Чтение пользовательской команды `RUN`.
2. В обработчике команды, осуществляется смена режима окружения на *выполнение программы*.
3. Выполнение программы, путем вызовов обработчиков инструкций кода.
4. Обработчик инструкции `INPUT` останавливает выполнение программы, поскольку требуется входные данные соответствующей инструкции. После ввода данных, необходимо продолжить выполнение программы (этап 3).
5. Завершение выполнения, смена режима окружения на *интерпретации команд*.

2.3 Среда разработки в графическом режиме

Графический режим среды исполнения позволяет интерпретировать графические команды GW-Basic (см. п. 1.1.3). В отличие от текстового режима, в графическом режиме требуется введение *холста* (англ. *Canvas*) – область, на которой отображается результат выполнения графической команды GW-Basic.

В тоже время, возможность ввода текста должна быть сохранена, поэтому необходимо ввести *матрицу текста*, которая будет отображать следующую информацию:

1. Пользовательский запрос в процессе ввода.

2. Результат выполнения неграфических команд.

В отличие от текстовой информации, графическая отображается только после выполнения соответствующей инструкции. Для исключения возможности перекрытия текста графикой, визуализацию кадра необходимо выполнять в следующей последовательности:

1. Отображение холста.
2. Отображения текущей матрицы текста.

3 Реализация среды GW-Basic

Реализация среды разработки GW-Basic осуществлялась на языке C. В результате, пользователю доступны исполняемые файлы со следующими режимами работы консоли:

1. Консоль текстового режима (исполнение только неграфических команд).
2. Консоль графического режима (исполнение текстовых и графических команд).

3.1 Лексический и синтаксический анализаторы

Генерация лексического анализатора осуществлялась с помощью инструмента *flex*.

За генерацию синтаксического анализатора отвечает инструмент *bison*. Краткое описание грамматики синтаксического разбора представлено в п. 2.1.2.

Поскольку целью выполнения этих разборов является построение дерева синтаксического разбора, то необходимо ввести *структуры языка C*, объекты которых будут являться узлами этого дерева в памяти программы. Данная реализация включает *уникальную структуру* для каждого нетерминала грамматики (см. листинг 4):

Листинг 4: Пример объявления структур для вершин AST-дерева

```
1 struct GWBN_Interpreter{
2     int type;          /* GWBNT_DIRECTMODE, GWBNT_INDIRECTMODE */
3     union {
4         struct GWBN_DirectMode* direct;
5         struct GWBN_IndirectMode* indirect;
6     };
7 };
8 struct GWBN_IndirectMode{
9     int line_number;
10    GWBN_Statements* statements;
11 };
12 struct GWBN_DirectMode {
13     /* Struct Fields*/
14 };
```

Исходя из такого подхода описания структур для каждого нетерминала грамматики (листинг 4), получается что деререво синтаксического разбора полностью укладывается в

корневой терминал **Interpreter** (см. листинг 1). Если дочерний узел дерева (нетерминал) не может быть определен однозначно, то для этих целей вводится поле **type** (например, строка 2, листинг 4).

3.2 Система времени выполнения

Одним из компонентом системы исполнения является *окружение*. Эта структура содержит информацию о введенных пользователем данных, и контексте выполнения. Объявление структуры представлено в листинге 5:

Листинг 5: Объявление структуры **Environment**, а также вложенных структур

```
1 struct GWBE_Environment {
2     int runtime_type;          /* GWBE_MODE_INTERPRETER, GWBE_MODE_PROGRAM */
3     struct GWBE_Context* ctx;
4     struct GWBE_Input* input;
5 };
6 struct GWBE_Input {
7     char* buffer;
8     size_t buffer_len;
9 };
10 struct GWBE_Context {
11     GWBE_Program* program;
12     GWBC_VariableListNode* system_vars, local_vars[];
13     GWBE_CallbackStack* callback_stack;
14 };
```

Для описания текущего режима времени исполнения используется поле *runtime_type* (строка 2, листинг 5), которое может принимать два значения (согласно п. 2.2.1). Введенные пользователем данные хранятся в структуре **GWBE_Input** (см. листинг 5).

Рассмотрим подробнее структуры, которые требуются для описания *контекста выполнения*. Согласно листингу 5, структура **GWBE_Context** включает в себя:

1. Текущую программу на GW-Basic.
2. Переменные (среды GW-Basic, пользовательские).
3. Стек возврата.

Поскольку любое взаимодействие со средой осуществляется с помощью пользовательских запросов, то и создание программы на GW-Basic происходит тоже с помощью запросов. За счет того, что любой запрос пользователя проходит этап обработки, вместе с текстом строк программы дополнительно хранятся вершины дерева синтаксического разбора (см. листинг 6). Такое решение позволяет не вызывать анализатор каждый раз при выполнении соответствующей инструкции, а сделать это один раз – только при добавлении/замене инструкции.

Листинг 6: Представление строк программы GW-Basic

```
struct GWBE_ProgramLine {
    int number;
    char* source;
    struct GWBN_Statements* stmts;
};
```

Что касается реализации такого компонента системы как *обработчиков вершин дерева синтаксического разбора*, то сигнатуры функций выглядят следующим образом (см. листинг 8):

Листинг 7: Пример сигнатуры обработчика вершины Interpreter

```
GWBR_Result gwbh_Interpreter(GWBE_Environment *env, GWBN_Interpreter* node)
{
    /* Implementation */
}
```

Для определения возникновения ошибки в процессе выполнения команд, в обработчиках предусмотрена структура `GWBR_Result`, которая содержит результат выполнения обработки. Таким образом, путем вставки проверок в каждый обработчик, можно получать стек вызовов в случае возникновения ошибки в процессе обработки вершины дерева.

3.3 Консоль графического режима

В основе реализации графического режима лежит использование инструментария *Glut*, и библиотеки *OpenGL*. Согласно п. 2.3, графический режим подразумевает отображение следующих объектов:

1. Отображение холста.

2. Отображение текущей матрицы текста.

Под холстом в данной реализации понимается матрица пикселей, на которую производится растеризация графических примитивов. Растеризация производится после выполнения очередной графической команды. Поскольку добавлять новые примитивы можно только в процессе построения кадра (т.е. в функции обратного вызова `glutDisplayFunc`), то отображение холста состоит из следующих этапов:

1. Загрузка холста в буфер кадра (`glWritePixels [2]`).
2. Визуализация примитива.
3. Чтение холста из буфера кадра (`glReadPixels [2]`).

Обновление информации в матрице текста осуществляется путем объявления *Glut* функции обратного вызова `glutKeyboardFunc`. Обработка нажатий происходит следующим образом (см. листинг 8):

1. При нажатии клавиши *backspace*, из матрицы текста *TextBuffer*, и из запроса пользователя удаляется последний символ (строки 2-3).
2. При нажатии *enter*, производится запуск среды исполнения, а после – очистка пользовательского запроса (строки 6-8).
3. При нажатии любого другого символа, происходит добавление символа в матрицу текста и строку запроса (строки 11-12).

Листинг 8: Обработка нажатий клавиш

```
1 case '\b': /* Backspace */
2     gwbg_TextBuffer_PopChar(ide->text_buffer);
3     gwbg_Environment_PopCharFromRequest(ide->env);
4     break;
5 case 13: /* Enter */
6     gwbg_TextBuffer_CursorNextLine(ide->text_buffer);
7     gwbr_Run(ide->env); /* Run user request */
8     gwbg_Environment_ClearRequest(ide->env); /* Clear user request */
9     break;
10 default: /* Another char */
```

```
11     gwbg_TextBuffer_PushChar(ide->text_buffer , key);  
12     gwbg_Environment_PushCharToRequest(ide->env , key);  
13     break;
```

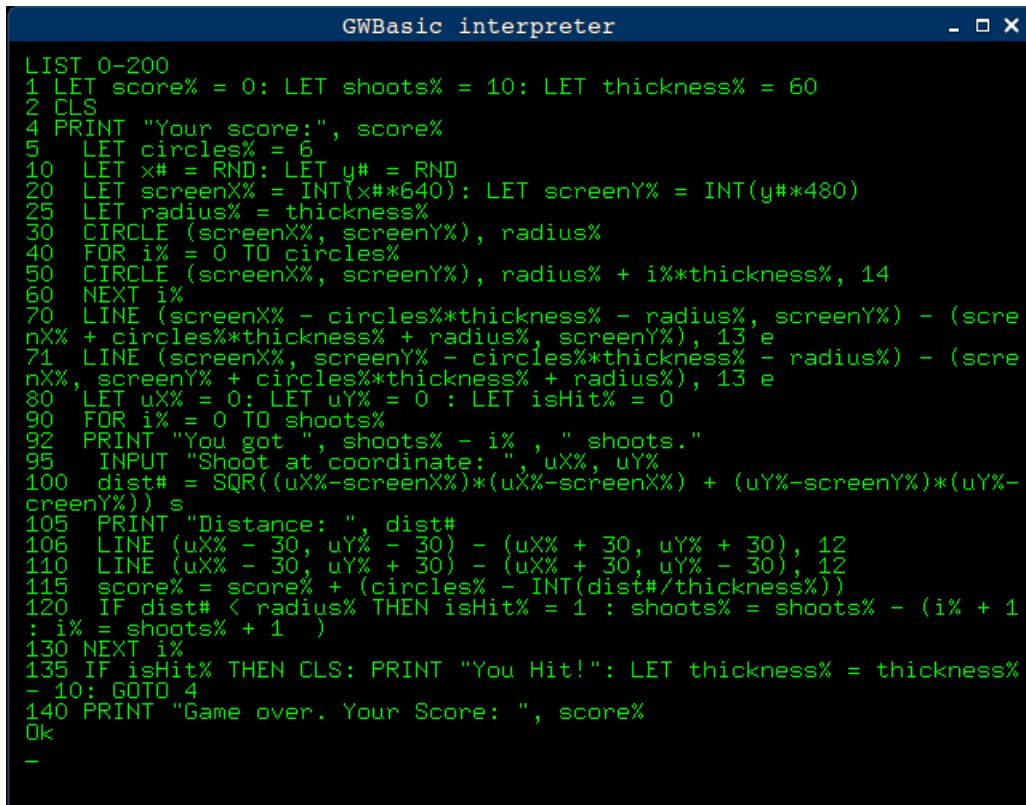
4 Тестирование

Тестирование разработанной среды GW-Basic заключается в выполнении программы, написанной на языке GW-Basic. В качестве таковой программы является игра "стрельба по мишени" исходный текст которой представлен в *Приложении А*.

4.1 Тестирование игры в графической консоли GW-Basic

Рассмотрим игровой процесс тестируемой игры. Игроку необходимо совершать выстрелы по мишеням, которые появляются в произвольном месте экрана. Новая мишень появляется в том случае, если игрок попал в центральное кольцо текущей. Для совершения выстрела, пользователю необходимо указать координаты стрельбы. Цель – набрать как можно больше очков.

Демонстрация работы консоли графического режима представлена на рисунках 2, 3:



```
GWBasic interpreter
LIST 0-200
1 LET score% = 0: LET shoots% = 10: LET thickness% = 60
2 CLS
4 PRINT "Your score:", score%
5 LET circles% = 6
10 LET x# = RND: LET y# = RND
20 LET screenX% = INT(x#*640): LET screenY% = INT(y#*480)
25 LET radius% = thickness%
30 CIRCLE (screenX%, screenY%), radius%
40 FOR i% = 0 TO circles%
50 CIRCLE (screenX%, screenY%), radius% + i%*thickness%, 14
60 NEXT i%
70 LINE (screenX% - circles%*thickness% - radius%, screenY%) - (screenX% + circles%*thickness% + radius%, screenY%), 13 e
71 LINE (screenX%, screenY% - circles%*thickness% - radius%) - (screenX%, screenY% + circles%*thickness% + radius%), 13 e
80 LET uX% = 0: LET uY% = 0: LET isHit% = 0
90 FOR i% = 0 TO shoots%
92 PRINT "You got ", shoots% - i%, " shoots."
95 INPUT "Shoot at coordinate: ", uX%, uY%
100 dist# = SQR((uX% - screenX%)*(uX% - screenX%) + (uY% - screenY%)*(uY% - screenY%))
105 PRINT "Distance: ", dist#
106 LINE (uX% - 30, uY% - 30) - (uX% + 30, uY% + 30), 12
110 LINE (uX% - 30, uY% + 30) - (uX% + 30, uY% - 30), 12
115 score% = score% + (circles% - INT(dist#/thickness%))
120 IF dist# < radius% THEN isHit% = 1: shoots% = shoots% - (i% + 1): i% = shoots% + 1
130 NEXT i%
135 IF isHit% THEN CLS: PRINT "You Hit!": LET thickness% = thickness% - 10: GOTO 4
140 PRINT "Game over. Your Score: ", score%
Ok
-
```

Рис. 2 – Консоль графического режима, демонстрация *прямого* режима.

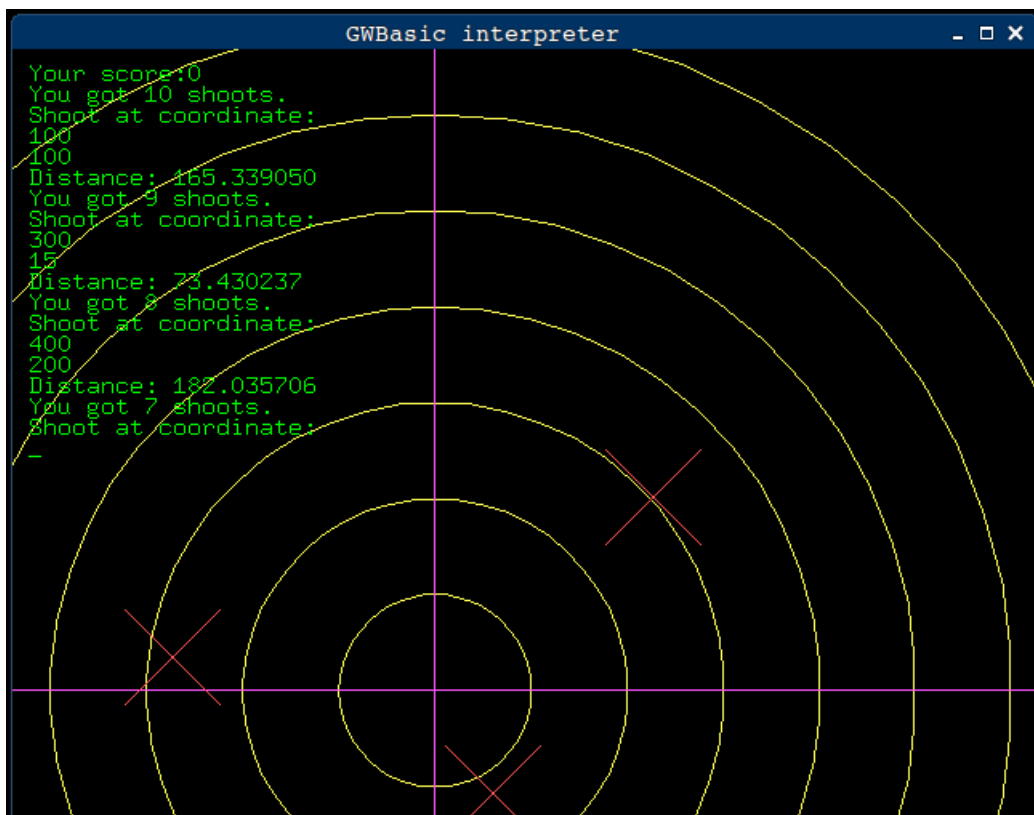


Рис. 3 – Консоль графического режима в процессе игры.

Заключение

В данной работе была рассмотрена реализации среды GW-Basic с использованием средств *flex*, *bison*, *OpenGL*, *glut* на языке *C*. Возможности среды можно легко расширить путем редактирования грамматики, а также добавлением обработчиков для соответствующих вершин дерева синтаксического разбора.

Список литературы

- [1] Glut callback registration [Электронный ресурс]. – Режим доступа: <https://www.opengl.org/documentation/specs/glut/spec3/node45.html>.
- [2] Opengl 2.1 reference page [Электронный ресурс]. – Режим доступа: <https://www.opengl.org/sdk/docs/man2/xhtml/>.
- [3] Gw-basic user's guide [Электронный ресурс]. – Режим доступа: <http://www.richardlemay.com/JEUX/GW-Basic/GW%20Basic%20User%20Guide.pdf>, 2004.
- [4] *John R. Levine.* flex & bison [Электронный ресурс]. – Режим доступа: http://web.iitd.ac.in/sumeet/flex_bison.pdf, 2009.
- [5] *Mark J. Kilgard.* The opengl utility toolkit (glut) programming interface api version 3 [Электронный ресурс]. – Режим доступа: <https://www.opengl.org/resources/libraries/glut/spec3/spec3.html>, 1996.

Содержание

Введение	1
1 Обзор используемых технологий	2
1.1 Среда разработки GW-Basic	2
1.1.1 Синтаксические конструкции	2
1.1.2 Режимы интерпретации запросов	3
1.1.3 Графический режим	3
1.2 Flex и Bison – инструменты разработки анализаторов	4
1.3 Glut – инструментарий для работы с OpenGL	4
2 Разработка среды GW-Basic	5
2.1 Анализаторы языка GW-Basic	5
2.1.1 Лексический анализатор	5
2.1.2 Синтаксический анализатор	5
2.2 Архитектура системы исполнения	7
2.2.1 Окружение среды	7
2.2.2 Обработка пользовательских запросов	8
2.2.3 Выполнение команд и программы	8
2.3 Среда разработки в графическом режиме	9
3 Реализация среды GW-Basic	11
3.1 Лексический и синтаксический анализаторы	11
3.2 Система времени выполнения	12
3.3 Консоль графического режима	13
4 Тестирование	16
4.1 Тестирование игры в графической консоли GW-Basic	16
Заключение	18

Приложение А. Тестовая программа на GW-Basic

```
1      LET score% = 0: LET shoots% = 10: LET thickness% = 60
2      CLS
4      PRINT "Your_score:", score%
5      LET circles% = 6
10     LET x# = RND: LET y# = RND
20     LET screenX% = INT(x#*320): LET screenY% = INT(y#*240)
25     LET radius% = thickness%
30     CIRCLE (screenX%, screenY%), radius%
40     FOR i% = 0 TO circles%
50         CIRCLE (screenX%, screenY%), radius% + i%*thickness%, 14
60     NEXT i%
70     LINE (screenX% - circles%*thickness% - radius%, screenY%) -
        (screenX% + circles%*thickness% + radius%, screenY%), 13
71     LINE (screenX%, screenY% - circles%*thickness% - radius%) -
        (screenX%, screenY% + circles%*thickness% + radius%), 13
80     LET uX% = 0: LET uY% = 0 : LET isHit% = 0
90     FOR i% = 0 TO shoots%
92         PRINT "You_got_", shoots% - i% , "_shoots."
95         INPUT "Shoot_at_coordinate:_", uX%, uY%
100        dist# = SQR((uX%-screenX%)*(uX%-screenX%) +
            (uY%-screenY%)*(uY%-screenY%))
105        PRINT "Distance:_", dist#
106        LINE (uX% - 30, uY% - 30) - (uX% + 30, uY% + 30), 12
110        LINE (uX% - 30, uY% + 30) - (uX% + 30, uY% - 30), 12
115        score% = score% + (circles% - INT(dist#/thickness%))
120        IF dist# < radius% THEN isHit% = 1 :
            shoots% = shoots% - (i% + 1): i% = shoots% + 1
130    NEXT i%
135    IF isHit% THEN CLS: PRINT "You_Hit!":
        LET thickness% = thickness% - 10: GOTO 4
140    PRINT "Game_over._Your_Score:_", score%
```