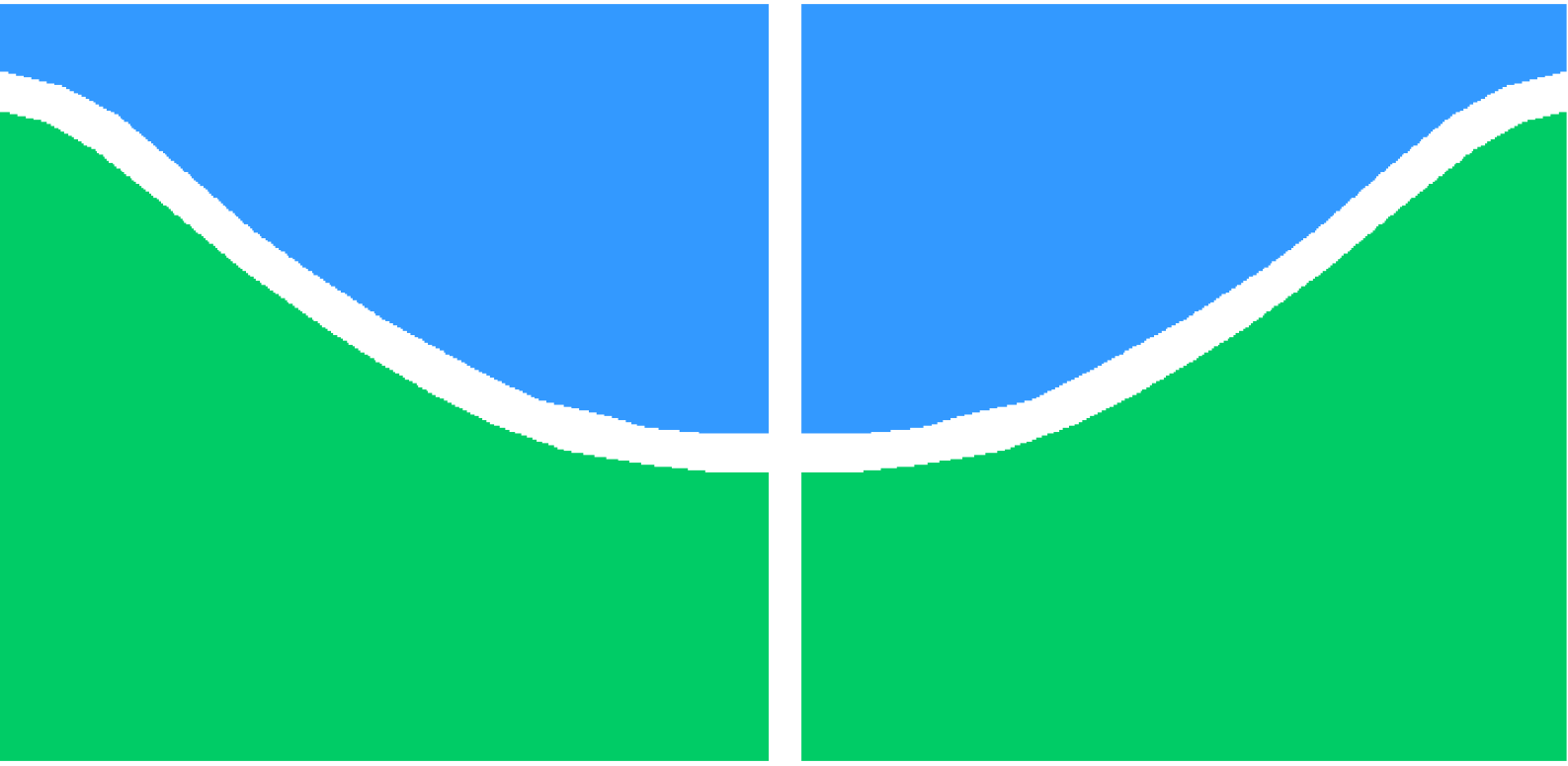

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia Eletrônica

Colli: uma linguagem de *script* para detecção de colisões em 2D

Autor: Guilherme de Oliveira Costa
Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF
2014

Guilherme de Oliveira Costa

Colli: uma linguagem de *script* para detecção de colisões em 2D

Monografia submetida ao curso de graduação em Engenharia Eletrônica da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Eletrônica.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF

2014

Guilherme de Oliveira Costa

Colli: uma linguagem de *script* para detecção de colisões em 2D/ Guilherme de Oliveira Costa. – Brasília, DF, 2014-

77 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2014.

1. Colisões. 2. Interpretadores. I. Prof. Dr. Edson Alves da Costa Júnior. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Colli: uma linguagem de *script* para detecção de colisões em 2D

CDU 02:141:005.6

Guilherme de Oliveira Costa

Colli: uma linguagem de *script* para detecção de colisões em 2D

Monografia submetida ao curso de graduação em Engenharia Eletrônica da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Eletrônica.

Trabalho aprovado. Brasília, DF, 01 de junho de 2013:

Prof. Dr. Edson Alves da Costa Júnior
Orientador

Prof. Dr. Marcelino Monteiro de Andrade
Convidado 1

Prof. Dr. Fábio Macêdo Mendes
Convidado 2

Brasília, DF
2014

À minha avó, que fazia o melhor torresmo do mundo.

Agradecimentos

Agradeço aos meus pais, Josi e Jalis, por todo amor e conselhos que me deram, bem como todos os livros comprados.

Ao meu irmão Henrique, às vezes chato, às vezes um grande amigo, e que certo dia apareceu com um tal de *video game* que mudou minha vida.

Aos barulhentos Kako, Hannah, Sid e Nina, sempre capazes de me fazer esquecer das preocupações.

Aos meus amigos que, morando na rua debaixo ou no outro lado do país, fazem parte de quem sou.

Aos professores Edson, Marcelino e Fábio, que me guiaram durante a graduação, e sem os quais não teria chegado tão longe.

"Video games fazem mal?
Foi isso que disseram do rock'n'roll."
(Shigeru Miyamoto)

Resumo

Este trabalho descreve o processo de criação de uma linguagem de *script* para detecção de colisões em simulações 2D, discutindo a maneira como esta foi implementada, desde da criação de seu conjunto sintático até os algortimos utilizados nos testes de colisão, bem como as ferramentas e técnicas utilizadas. O texto também apresenta o ferramental de apoio desenvolvido para ser utilizado em conjunto com a linguagem, consistindo em um módulo de visualização e uma API de integração com a linguagem C.

Palavras-chaves: colisões. interpretador. *parsing*.

Abstract

This document describes the creation process of a script language for the detection of collisions in 2D computer simulations, discussing the implementation decisions for this language, touching on aspects such as its syntax and collision testing algorithms, as well as the tools and techniques used on its development. The text also discusses the development process for the auxiliary tools of the language, such as a visualisation module and an API for integration with the C language.

Key-words: collisions. interpreter. parsing.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

Uma parte vital do desenvolvimento de jogos é a detecção de colisões, que consiste em analisar a disposição espacial de objetos geométricos descritos computacionalmente e determinar se ocorre superposição entre os mesmos. Usualmente, os sistemas de colisão em jogos devem apenas "parecer corretos", pois erros de colisão de poucos *pixels* não afetam significativamente um jogo. Porém, um sistema de colisão com erros em demasia quebra totalmente a imersão do jogador, levando a situações incomuns, como mostrado na Figura 1.

Figura 1 – Colisão incorreta no jogo *Big Rigs: Over The Road Racing*.

Entretanto, trabalhar com colisões pode ser demasiadamente complexo e custoso, já que as soluções atuais ou são codificadas diretamente na estrutura do jogo, sendo implementadas a partir do zero, ou utilizam soluções já prontas, como por exemplo as disponíveis em Unity[1]. Estas aproximações são complexas demais (caso o programador decida implementar seu próprio sistema), ou firmemente atreladas à *engine* de física da ferramenta (como os *Rigidbodies* em Unity).

Este documento apresenta Colli, uma linguagem de *script* para descrição e teste de colisão em simulações bidimensionais, onde o usuário deve utilizar as expressões da linguagem para criar uma representação do estado atual dos objetos em sua simulação

[1] http://unity3d.com/

e requisitar a realização dos testes de colisão nos objetos de seu interesse. Esta abordagem visa deixar a detecção de colisão transparente para o desenvolvedor, possibilitando que este dedique sua atenção à implementação da *engine* de física, tirando o foco da implementação do sistema de detecção de colisão e facilitando o aprendizado. Além disso foram desenvolvidas ferramentas de apoio com o objetivo de auxiliar os desenvolvedores que optarem por utilizar Colli, como um módulo de visualização e uma API (*Application Programming Interface*) de integração com a Linguagem C.

Parte integral do desenvolvimento do trabalho consistiu então do projeto da linguagem, incluindo a descrição EBNF e manual da mesma, bem como a implementação em código de cada *feature* proposta. Buscou-se através deste esforços, a criação de uma linguagem e de ferramentas simples e objetivas, com resultados facilmente reprodutíveis.

Este documento apresenta primeiramente uma discussão sobre as referências teóricas escolhidas para auxiliar o desenvolvimento do trabalho. Estas referências dizem respeito à teoria de colisões, compiladores e às ferramentas e técnicas utilizadas no desenvolvimento deste tipo de solução. Após esta seção são discutidos então os passos para o desenvolvimento do mesmo, bem como a reprodução dos resultados obtidos, e uma discussão dos mesmos.

2 Fundamentação Teórica

2.1 Linguagens e Interpretadores

De acordo com Aho et al. (2013), um compilador é "um programa que recebe como entrada um programa em uma linguagem de programação – a linguagem *fonte* – e o traduz para um programa equivalente em outra linguagem – a linguagem *objeto*". Dessa forma, um compilador realiza a tradução de uma linguagem para outra, gerando um código objeto. Este código objeto pode ser executado, recebendo entradas e gerando saídas. Um interpretador, por sua vez, é caracterizado por seguir diretamente para a etapa de execução, recebendo o código fonte e as entradas e gerando as saídas, sem criar um programa intermediário. Essas relações são mostradas na Figura 2.

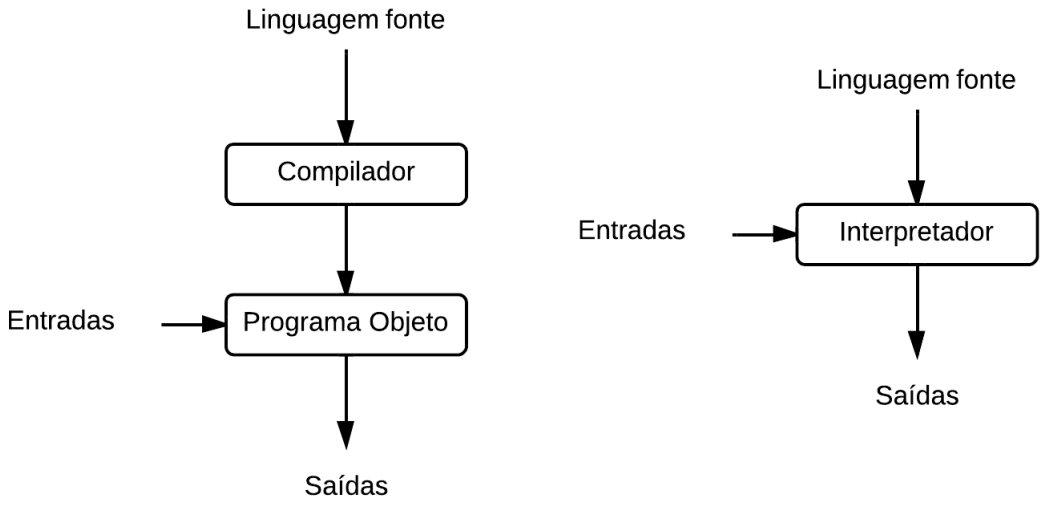

Figura 2 – Diagrama ilustrando a estrutura de um compilador e de um interpretador.

Em ambos os casos o procedimento é realizado em duas etapas: a análise léxica e a análise semântica (*parsing*). A primeira etapa analisa os blocos básicos que constituem uma linguagem (lexemas): as informações coletadas nesse passo são então analisadas pelo *parser*, que verifica se os comandos do código fonte são válidos. A diferença entre um compilador e um interpretador está no resultado da análise sintática: um compilador gera um código em outra linguagem intermediária (*Assembly* ou C, por exemplo) e o interpretador executa os comandos imediatamente. Linguagens interpretadas usualmente

são referidas como linguagens de *script*[1].

É importante notar que a análise léxica verifica se as palavras digitadas fazem parte da gramática da linguagem, sem analisar se essas palavras fazem sentido. Um exemplo clássico é a frase "O sofá pulou no gato" – enquanto todos os lexemas utilizados nessa frase são válidos quanto à língua portuguesa, a semântica é falha, pois sofás são objetos inanimados. Basicamente, a análise léxica verifica a "ortografia" da frase e a análise semântica diz respeito ao sentido da frase.

Usualmente os lexemas contém as *palavras reservadas*, isto é, termos que não podem ser utilizados para nomear identificadores no contexto da linguagem. Exemplos disso são os termos `int` e `double`, utilizados para definição do tipo de certos elementos em várias linguagens de programação.

Os lexemas surgem da descrição da linguagem em uma gramática livre de contexto, que consiste em um conjunto de não terminais, terminais, e regras de produção (AHO et al., 2013):

- **Não terminais**: são variáveis sintáticas que representam conjuntos de cadeias (que podem ser constituidas de outros não terminais e/ou terminais).
- **Terminais**: são os símbolos básicos da linguagem e que não podem ser quebrados em cadeias. Usualmente são os lexemas da linguagem.
- **Regras de Produção**: descrição de como os não terminais e terminais devem ser dispostos para formar uma expressão válida.

O uso de uma gramática livre de contexto permite utilizar recursão para definir uma expressão, isto é, utilizar uma mesma regra para descrever uma expressão com n componentes. Um exemplo seria a definição recursiva dos elementos em uma lista, que pode conter um elemento apenas ou uma lista seguida de um elemento. Esta lista, por sua vez, pode conter um elemento ou uma lista seguida de outro elemento, e assim sucessivamente. Uma forma muito popular para se descrever linguagens de programação é o BNF (*Backus Naur Form*) e suas derivadas. Ao longo deste trabalho foi utilizado o padrão EBNF (*Extended Backus Naur Form*), descrito no anexo A.1 de W3C (2014). O exemplo da lista é mostrado em EBNF no Código 2.1 (para este exemplo, considere uma lista de números de 0 a 9).

Código 2.1 – Exemplo de EBNF.

```
1 elemento ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
2 lista ::= (lista elemento) | elemento
```

[1] Exemplos de linguagens de *script* *LUA*, *Python* e *Matlab*.

Neste exemplo, os dígitos (0, 1, 2...) são os terminais, isto é, não podem ser reduzidos por nenhuma regra. Já `lista` e `elemento` são não-terminais, sendo uma combinação de regras.

Outro aspecto que deve ser definido no desenvolvimento uma linguagem de programação é a tipagem da mesma. Em memória, as variáveis são representadas como *bits*, cabendo ao compilador definir como aqueles dados serão interpretados, o que depende do tipo da variável. Existem duas aproximações para a tipagem (AHO et al., 2013): Tipagem Estática, onde o tipo de cada variável é definido previamente, sendo resolvido em tempo de compilação, e a Tipagem Dinâmica, onde o tipo de cada variável é deduzido em tempo de execução, podendo inclusive ser alterado. Usualmente, linguagens de *script* utilizam tipagem dinâmica.

Especialmente devido à maneira como a tipagem é tratada, linguagens de *script* possuem um nível maior de abstração, isto é, os comandos são mais compactos e ocultam parte da complexidade envolvida na execução de suas operações. A título de exemplo, são apresentados dois códigos equivalentes, um escrito em C (Código 2.2) e outro escrito em *Python* (Código 2.3). O problema a ser resolvido é ler um número escrito por um usuário (armazenado na variável `num`) e imprimir na tela o dobro deste número.

Código 2.2 – Exemplo em C.

```c
#include <stdio.h>

int main(){

        int num;

        scanf("%d", &num);
        printf("%d\n", 2*num);

        return 0;
}
```

Por se tratar de uma linguagem estáticamente tipada, C necessita que o a variável a ser utilizada para armazenar o resultado seja declarada préviamente, assim como o tipo esperado para leitura e escrita seja especificado (`%d` representando um número inteiro na base decimal, utilizado nas funções **`scanf()`** e **`printf()`**).

Código 2.3 – Exemplo em Python.

```python
num = input()
print(2*num)
```

Já a implementação em *Python* da solução decide automáticamente o tipo da variável digitada pelo usuário, produzindo em seguida a saída do programa. Para as entradas do tipo inteiro, ambos os programas se comportam de maneira similar, porém, ao inserir um número real como entrada o comportamento dos programas é diferente. A solução escrita em C irá *truncar* o valor lido, isto é, caso o usuário digite os valores 1, 1.8 ou 1.3, o valor obtido será a parte inteira destes números. Em *Python*, o tipo de variável será determinado corretamente e o resultado obtido será o esperado.

2.2 Colisões

2.2.1 Princípios Básicos

De acordo com Ericson (2004), um bom sistema de colisões deve responder a três questões:

1. **Se** os objetos em cena estão colidindo;
2. **Quando** estes objetos colidem;
3. **Onde** a colisão ocorre.

Para a realização dos testes de colisão, é necessário definir quais formas serão colocadas sob análise. Em um primeiro momento pode-se considerar realizar os testes diretamente na geometria de renderização[2], mas esta é, de forma geral, extremamente complexa, levando a testes muito custosos. Uma outra aproximação consiste em envolver os objetos em formas geométricas mais simples, denominados *bounding boxes*, e então realizar os testes nestas formas geométricas. Na Figura 3 é possível ver o *bounding box*, que representa a personagem do jogo, cuja geometria de renderização é cheia de curvas e formas complexas.

Um *bounding box* pode ser representado por conjunto S de pontos interligados entre si, representando uma forma geométrica. O conjunto S pode configurar tanto uma geometria convexa (na qual os ângulos internos do polígono formado são todos menores do que 180 graus) ou côncava (onde pelo menos um ângulo interno é maior que 180 graus). Um vértice com ângulo interno maior que 180 graus é chamado de vértice côncavo, sendo o vértice convexo seu recíproco. Outra forma de definir convexidade é considerar que para um polígono ser convexo, todo e qualquer segmento de linha entre os pontos de S deve estar contido dentro do polígono delimitado por ele. A Figura 4 ilustra estes conceitos.

Idealmente os *bounding boxes* devem ser ajustar bem à geometria de renderização, ocupar pouco espaço em memória e serem eficientes quanto aos testes que podem ser reali-

[2] A geometria de renderização é a geometria apresentada ao usuário.

Figura 3 – *Bounding Box* envolvendo a personagem do jogo Dauphine.

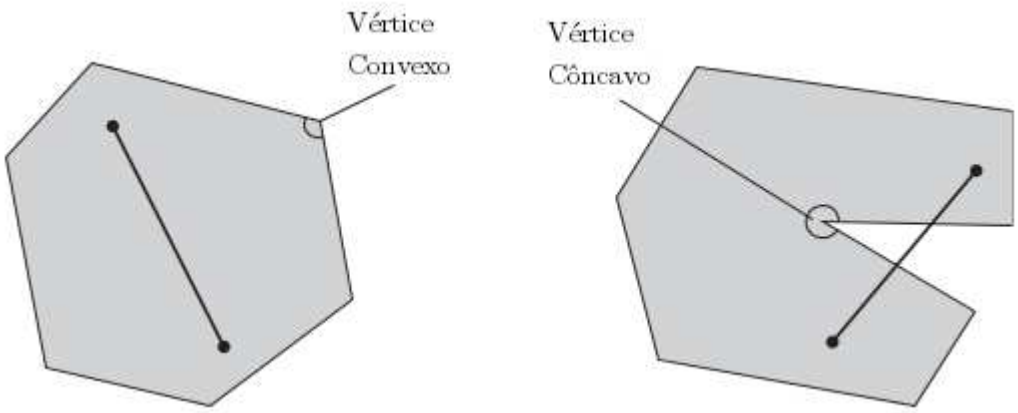

Figura 4 – Convexidade de polígonos. Adaptado de Ericson (2004).

zados sobre eles. Porém aliar estas características é uma tarefa extremamente difícil, pois geometrias complexas utilizam bastante espaço em memória, e nem sempre configuram uma representação geométrica convexa, aumentando a complexidade dos testes.

Dentre os tipos mais comuns de *bounding boxes* (mostrados na Figura 5), a esfera é mais simples de ser representado, necessitando apenas da localização do centro e o raio da mesma. O AABB (do inglês *Axis Aligned Bounding Box*) necessita de um ponto do polígono, assim como a largura e altura. *Bounding boxes* do tipo OBB (*Oriented Bounding Box*) são definidos por eixos x, y e z locais, podendo então ser representados como AABBs alinhados a este eixo local. Os k-DOPs (*Discrete Oriented Polytopes*) são represetados por orientações em k direções e necessitando apenas das distâncias mínimas e máximas em cada direção. Na Figura é mostrado um 8-DOP.

Os *convex hulls* são casos especiais, sendo computados como um conjunto mínimo C que constitui uma forma convexa entre P pontos. Dessa forma, alguns pontos de P

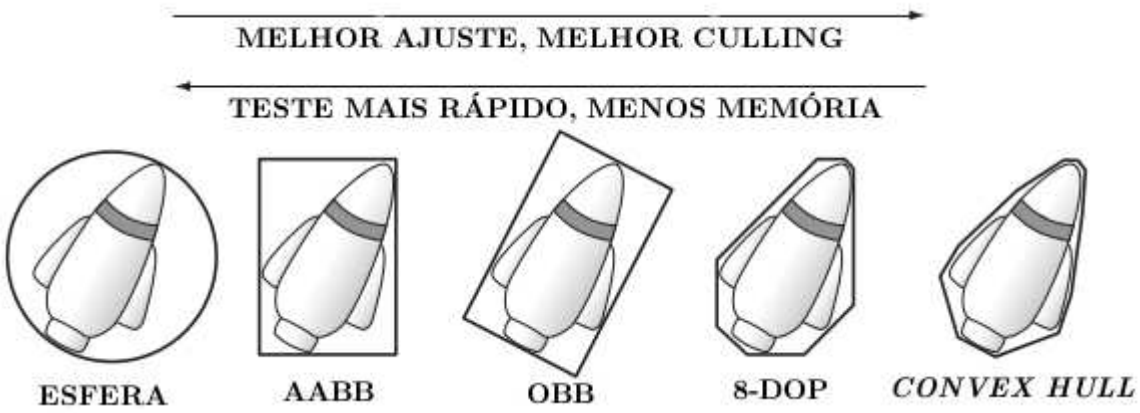

Figura 5 – Tipos mais comuns de *bounding boxes*. Adaptado de Ericson (2004).

podem não serem necessários para a representação do invólucro convexo. *Convex Hulls* podem ser obtidos através da aplicação do algoritmo de Andrew (ANDREW, 1979) ou do algoritmo *Quickhull* (WESTHOFF, 2014).

2.2.2 Testes Computacionais de Colisão

Uma vez descritos os *bounding boxes*, a representação da geometria de colisão se torna mais simples, porém os testes para detecção de colisão são não-triviais até para formas simplificadas, utilizando aplicações de álgebra linear e computação de alto desempenho.

2.2.2.1 Colisão entre Círculos

O teste de colisão mais simples possível é para dois objetos com *bounding boxes* circulares, sendo necessário apenas analisar a localização do centro geométrico de cada círculo e seus raios. Os objetos colidem caso seus raios estejam se sobrepondo. Um algoritmo para teste de colisão é mostrado a seguir:

Código 2.4 – Teste de colisão entre círculos.

```
1  typedef struct Circle_{
2          int x, y;
3          int radius;
4  } Circle;
5
6  int circlesColliding(Circle* c1, Circle* c2) {
7      int dx = c1->x - c2->x;
8      int dy = c1->y - c2->y;
9
10     int radii = c1->radius + c2->radius;
```

```
11
12      if ((dx * dx) + (dy * dy) < radii * radii)
13          return true;
14
15      return false;
16  }
```

De maneira geral, a colisão entre dois objetos só ocorre caso exista sobreposição tanto no eixo x quanto no eixo y, portanto a medida que as geometrias dos *bounding boxes* se tornam mais complexas, mais custosos são os testes a serem realizados.

2.2.2.2 Teste Genérico para Polígonos Convexos

Uma abordagem bastante utilizada em testes de colisão entre objetos de formas arbitrárias é o Teorema do Eixo de Separação (do inglês *Separating Axis Theorem*, ou SAT) (ERICSON, 2004). Este teste segue do Teste do Hiperplano[3] de Separação, que afirma que, dados dois conjuntos convexos A e B, ou este conjuntos se intersectam, ou eles estão separados por um hiperplano P. Dessa forma, segue que existe uma linha L perpendicular a P, que pode ser utilizada para avaliar a existência de superposição entre A e B. Este teste usa o fato de que polígonos convexos não podem "se enrolar" uns nos outros.

Para polígonos simétricos este teste assemelha-se ao teste para a colisão entre círculos: são analisados os segmentos de reta entre o centro geométrico de cada polígono e cada um de seus vértices. A projeção destes vértices (ilustrado pela Figura 6) em P e L é então avaliada para verificar a existência de colisão.

Eixos de Separação são facilmente encontrados por inspeção, porém estes são efetivamente infinitos, sendo desejável limitar o número de eixos a serem testados como candidatos a Eixos de Separação. Isto pode ser realizado escolhendo-se como candidatos aqueles eixos paralelos às faces dos polígonos de teste, bem como suas normais[4]. Essa escolha diz respeito à maneira como os objetos podem entrar em contato de acordo com sua geometria. Porém tais testes se tornam demasiadamente custosos para polígonos de várias faces. Por exemplo, num teste de colisão entre um polígono de 12 lados contra outro de 6 lados, seriam realizados testes em 36 direções diferentes. A Figura 7 mostra as projeções dos vértices de dois polígonos em várias direções de teste.

Entretanto é desejável decidir um conjunto de direções padrão para verificação de colisão. No desenvolvimento da linguagem Colli utilizou-se 12 direções (o intervalo de 0

[3] Um hiperplano é o nome dado a um plano em dimensões arbitrárias, possuindo uma dimensão a menos do que o espaço ao qual pertence (ERICSON, 2004). Em duas dimensões, um hiperplano consiste em uma linha.

[4] Ericson (2004) propõe que sejam realizados testes também em relação ao produto externo entre as faces de cada polígono, porém este caso só se aplica em três dimensões.

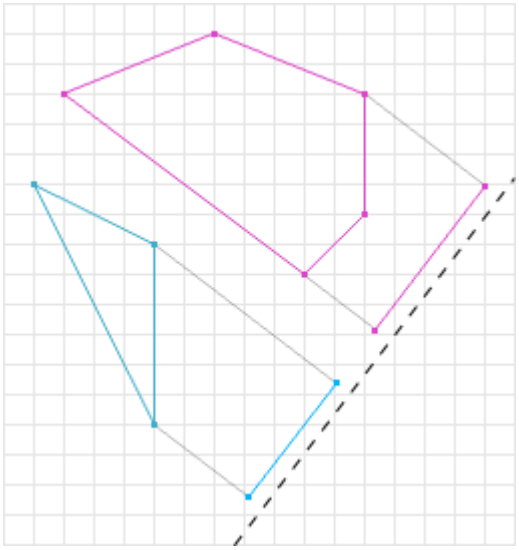

Figura 6 – Polígonos com suas projeções em uma reta (BITTLE, 2014).

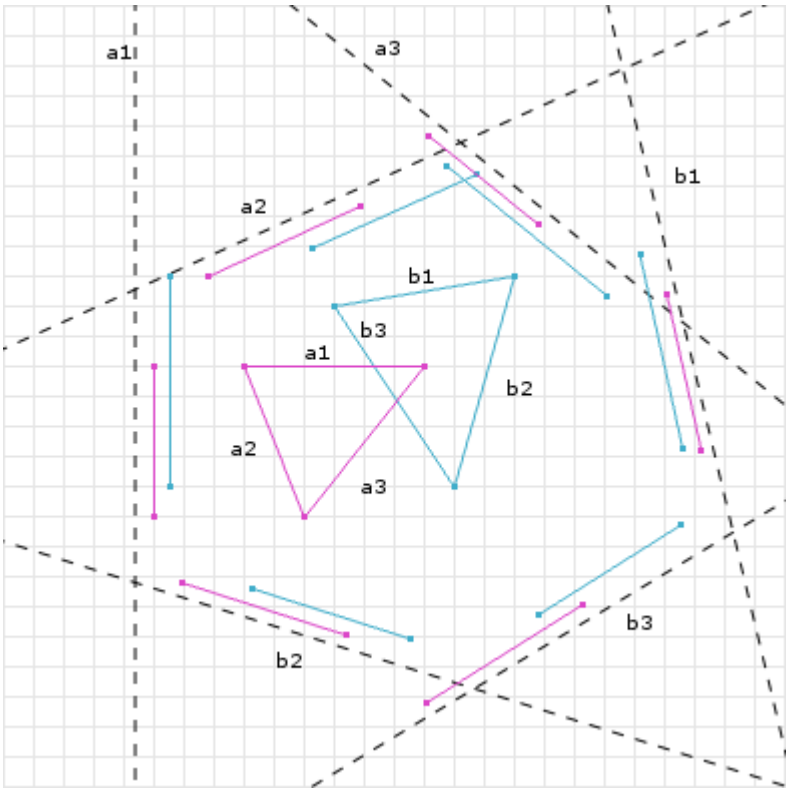

Figura 7 – Projeções dos vértices para realização do SAT (BITTLE, 2014).

a $\frac{\pi}{2}$, dividido em 6 partes uniformemente espaçadas, e suas retas normais). Com base nas projeções de todos os vetores na reta de teste, encontra-se a maior entre as menores projeções dos vértices ($maxMin$) de ambos os polígonos e a menor projeção entre as maiores ($minMax$). Caso $minMax - maxMin < 0$, isso implica que $maxMin$ projeção dos vértices de um dos polígonos está a frente de $minMax$, portanto não ocorre colisão. Uma vez que há separação em um eixo, o programa reporta que não há colisão, não

necessitando testar os vértices remanescentes.

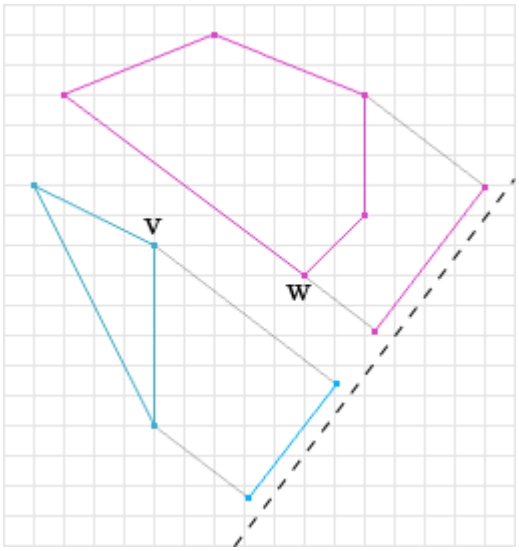

Figura 8 – Vértices $maxMin$ (**v**) e $minMax$ (**w**). Adaptado de (BITTLE, 2014).

Na Figura 8 os pontos $minMax$ e $maxMin$ (**v** e **w**, respectivamente) são mostrados. Por inspeção é possível ver que $maxMin > minMax$, não resultando em colisão. O Código 2.5 mostra a implementação em C do teste de colisão entre dois polígonos convexos.

Código 2.5 – Teste para colisão entre polígonos convexos arbitrários.

```
1  char polyCollision(doubleVector *A, doubleVector *B){
2
3          int i;
4          int j;
5
6          double u_x[2] = {1.0, 0.0};
7          double directions[12][2]; // Test directions
8
9          for (i = 0; i < 12; i++){
10                 double *rotated = rotate(u_x, i*PI/12);;
11
12                 directions[i][0] = rotated[0];
13                 directions[i][1] = rotated[1];
14         }
15
16         for (i = 0; i < 12; i += 1){
17
18                 double u[2];
```

```
19
20              u[0] = directions[i][0];
21              u[1] = directions[i][1];
22
23              double pt[2] = {0.0 , 0.0};
24
25              double *A_coords = (double *)malloc((A->size/2)*sizeof(double));
26              double *B_coords = (double *)malloc((B->size/2)*sizeof(double));
27
28              for(j = 0; j < A->size; j+=2){
29
30                      pt[0] = A->vectorElems[j];
31                      pt[1] = A->vectorElems[j + 1];
32
33                      //Scalar projection of a vertex onto the test axis
34                      A_coords[j/2] = dotProduct(pt, u);
35              }
36
37              for(j = 0; j < B->size; j+=2){
38
39                      pt[0] = B->vectorElems[j];
40                      pt[1] = B->vectorElems[j + 1];
41
42                      //Scalar projection of a vertex onto the test axis
43                      B_coords[j/2] = dotProduct(pt, u);
44              }
45
46              double Amax = A_coords[0];
47              double Amin = A_coords[0];
48
49              for(j = 0; j < A->size/2; j++){
50
51                      if(A_coords[j] > Amax)
52                              Amax = A_coords[j];
53
54                      if(A_coords[j] < Amin)
55                              Amin = A_coords[j];
56              }
57
```

```
58                double Bmax = B_coords[0];
59                double Bmin = B_coords[0];
60
61                for(j = 0; j < B->size/2; j++){
62
63                        if(B_coords[j] > Bmax)
64                                Bmax = B_coords[j];
65
66                        if(B_coords[j] < Bmin)
67                                Bmin = B_coords[j];
68                }
69
70                double minmax;
71
72                if(Amax < Bmax)
73                        minmax = Amax;
74                else
75                        minmax = Bmax;
76
77                double maxmin;
78
79                if(Amin > Bmin)
80                        maxmin = Amin;
81                else
82                        maxmin = Bmin;
83
84                double shadow = minmax - maxmin;
85
86                if (shadow < 0)
87                        return NO_COLLISION;
88        }
89
90        return COLLISION;
91 }
```

2.2.3 Teste entre Círculos e Polígonos

Quando deseja-se realizar um teste entre um polígono convexo qualquer e um círculo, utiliza-se um método levemente simplificado do teste de colisão entre polígonos.

Esta simplificação vem do fato de que o segmento de reta entre o centro geométrico de um círculo de raio R e o ponto deste círculo que mais se aproxima do polígono (ponto de suporte do círculo) tem tamanho R e fica na direção do segmento de reta entre os centros dos objetos. Portanto resta apenas encontrar o ponto de suporte do polígono, realizado de maneira similar ao que é feito no teste entre polígonos. Em relação à direção de teste, utiliza-se apenas a direção do segmento entre os centros dos objetos, e sua normal, o que se deve à geometria do círculo e como esta pode colidir com o polígono.

Neste caso, a separação em um dos eixos irá ocorrer caso a distância d entre os centros for maior que a projeção dos raios (segmento de reta entre os pontos de suporte e o centro do mesmo objeto) naquele sentido. A Figura 9 ilustra este teste.

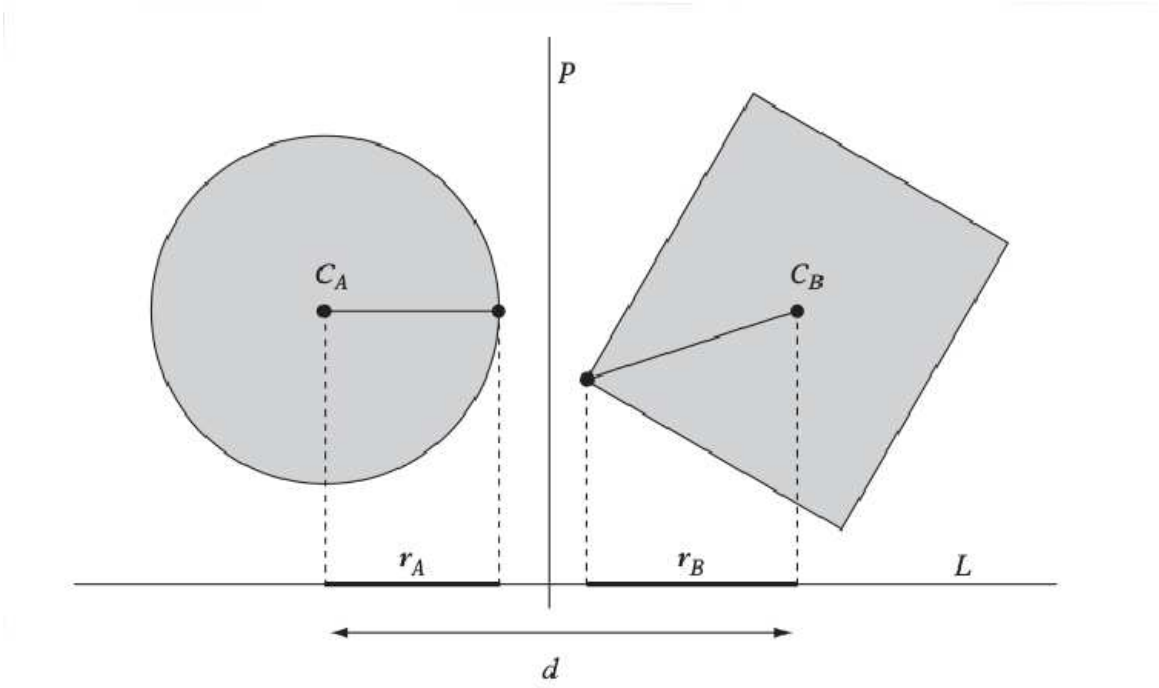

Figura 9 – Teste de Colisão entre Círculo e Polígono. Adaptado de Ericson (2004).

A seguir, o Código 2.6 mostra a maneira como este algoritmo foi implementado.

Código 2.6 – Teste para colisão entre polígonos convexos arbitrários.

```
1 char circlePoly(doubleVector *A, doubleVector *B){
2
3       //Assuming A is Circle, and B is Convex
4
5       int i;
6       int j;
7
8       double pt[2] = {0.0, 0.0};
9
```

```
10        double *polyProjections =
11                (double *)malloc((B->size/2)*sizeof(double));
12
13        pt[0] = A->centerX - B->centerX;
14        pt[1] = A->centerY - B->centerY;
15
16        double normalized[2];
17        normalized[0] = pt[0]/norm(pt);
18        normalized[1] = pt[1]/norm(pt);
19
20        double *sizedR = scalarMult(A->vectorElems[0], normalized);
21
22        double closestA[2]; //Closest point of A to B
23
24        closestA[0] = -sizedR[0] + A->centerX;
25        closestA[1] = -sizedR[1] + A->centerY;
26
27        for(j = 0; j < B->size; j+=2){
28
29                pt[0] = B->vectorElems[j];
30                pt[1] = B->vectorElems[j + 1];
31
32                //Scalar projection of a vertex onto
33                //the normalized center-to-center vector
34                polyProjections[j/2] = dotProduct(pt, normalized);
35        }
36
37        double closestB[2];
38
39        double Bmax = polyProjections[0];
40
41        for(i = 0; i < B->size/2; i++){
42
43                if(polyProjections[i] >= Bmax){
44
45                        Bmax = polyProjections[i];
46
47                        closestB[0] = B->vectorElems[2*i];
48                        closestB[1] = B->vectorElems[2*i + 1];
```

```
49                }
50        }
51
52        double centerA[2] = {A->centerX, A->centerY};
53        double centerB[2] = {B->centerX, B->centerY};
54
55        double *rA = vectorSub(closestA, centerA);
56        double *rB = vectorSub(closestB, centerB);
57
58        char result = COLLISION;
59
60        if(abs(rA[0]) + abs(rB[0]) < abs(centerA[0] - centerB[0]))
61                result = NO_COLLISION;
62
63        else if(abs(rA[1]) + abs(rB[1]) < abs(centerA[1] - centerB[1]))
64                result = NO_COLLISION;
65
66        return result;
67 }
```

2.3 Programação Paralela

2.3.1 Processos *Multithread*

Um ambiente computacional é caracterizado por uma constante competição por recursos, e é papel do sistema operacional garantir que estes recursos sejam compartilhados de forma a garantir o correto funcionamento dos processos em execução. De forma geral o sistema operacional reserva uma porção da memória para cada processo e toma medidas para que estes não sobrescrevam o espaço reservado um do outro.

Cada um destes processos pode ainda ser divido em *threads*. *Threads* podem ser vistas como subprocessos que compartilham o espaço de endereçamento de um processo maior (TANENBAUM, 2010). Isso implica que uma *thread* tem conhecimento de funções e variáveis globais de seu processo. Em geral, *threads* são utilizadas para parelelizar operações "lentas" (de um ponto de vista computacional), como tratamento de Entrada e Saída em processos.

O padrão de *threads* utilizado em sistemas Unix é o POSIX (TANENBAUM, 2010), desenvolvido como uma ferramenta para tornar portáveis os códigos *multithread*, já que cada fabricante de *hardware* realizava sua própria implementação de *threads*, dificultando

o processo de portabilidade de código entre diferente máquinas (BARNEY, 2014). *Threads* implementadas de acordo com o padrão POSIX são chamadas de *Pthreads*.

2.3.2 Comunicação Entre Processos

Por muitas vezes é desejável estabelecer um canal de comunicação entre processos que competem, de maneira que possam trocar informações. Estes mecanismos de Comunicação Entre Processos ou IPC (do inglês, *Inter Process Communication*) existem de diversas formas, sendo de especial interesse para este trabalho os *Pipes* e os Semáforos.

Também chamados de FIFOs (devido à maneira como tratam os dados, que segue o sistema *First In First Out*), os *Pipes* são canais de comunicação que ligam a saída de um processo a entrada de outro (GOLDT et al., 2014), como mostrado na Figura 10. O *pipe* é um dos IPCs mais antigos, e está presente na maior parte das distribuições *Unix*.

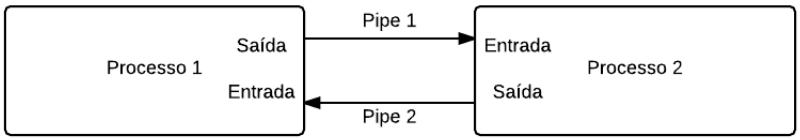

Figura 10 – Ilustração da comunicação através de *pipes*.

Os semáforos consistem em operações atômicas que controlam o acesso a um recurso, efetivamente realizando uma exclusão mútua (GOLDT et al., 2014). Nesta forma os semáforos são utilizados para proteger o acesso a um recurso crítico, isto é, um recurso cuja disponibilidade é essencial para execução de um processo (por exemplo, um semáforo que regula um serviço de impressão, de maneira a evitar que pedidos se misturem). Semáforos são utilizados também para sincronizar a execução de processos, o que é feito bloqueando o acesso a uma parte de uma rotina até que esta esteja pronta para ser realizada (por exemplo, um semáforo pode ser utilizado para bloquear a leitura de um arquivo até que este tenha alguma informações disponível).

2.4 Conceitos de Geometria

2.4.1 Transformações Lineares e Translação

Para um sistema de colisão abrangente, é desejável que este seja capaz de realizar certas operações, como rotação em torno do centro geométrico e expansão ou contração linear de um vetor. Tais operações caem dentro da categoria de *Transformações Lineares*. Por definição, uma Transformação Linear T de um vetor v consiste em multiplicar v por uma matriz A adequada, onde A é chamada de matriz de transformação (STRANG, 2009). Dessa forma, uma transformação linear pode ser expressa da seguinte maneira:

$$T(v) = Av$$

Considerando um vetor de coordenadas (x, y), a matriz de transformação, dada pela Equação 2.1 rotaciona este vetor em um ângulo θ no sentido anti-horário, até as coordenadas (x', y').

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & \text{sen}\theta \\ -\text{sen}\theta & \cos\theta \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} \tag{2.1}$$

Similarmente, a matriz para realizar a expansão ou contração linear de um vetor por um fator α, a transformação linear é dada pela Equação 2.2 (BOLDRINI et al., 1980), sendo que as expansões são caracterizadas por valores $\alpha > 1$ e as contrações por $0 < \alpha < 1$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \alpha & 0 \\ 0 & \alpha \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} \tag{2.2}$$

Para realizar estas transformações em um polígono, basta aplicar cada transformação sucessivamente a cada um de seus vértices. Entretanto, estas transformações levam em consideração que o polígono está centrado na origem do sistema de coordenadas. Para realizar tais operações corretamente, é necessário centrar o objeto na origem, aplicar transformação desejada, e então voltar com o objeto para sua posição original.

A translação é um caso à parte, não sendo possível representar a mesma através de uma transformação linear no mesmo número de dimensões do vetor o qual deseja-se transladar[5], sendo necessário realizar tal operação adicionando uma dimensão extra. Para realização de tal operação no mesmo número de dimensões do vetor, basta somar um *offset* ao vetor, como mostrado na Equação 2.3.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x_{offset} \\ y_{offset} \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix} \tag{2.3}$$

2.4.2 Centro Geométrico de um Objeto

Por definição, o centro geométrico de um polígono é um ponto interior ao mesmo cujas coordenadas são dadas pelo primeiro momento dos pontos que definem o polígono (M_x e M_y), dividido por sua área A (THOMAS et al., 2008). As Equações 2.4, 2.5 e 2.6 mostram como estes valores podem ser calculados.

$$A = \iint dxdy \tag{2.4}$$

[5] A translação falha em um dos testes de transformação linear, no qual a transformação do vetor nulo $(0, 0)$ deve resultar no vetor nulo $(0, 0)$.

$$M_x = \iint y dx dy \tag{2.5}$$

$$M_y = \iint x dx dy \tag{2.6}$$

Desta forma, o centro de massa é posicionado nas x ($coordenadas_{CM}$, y_{CM}), e cujos valores são dados pelas Equações 2.7 e 2.8:

$$x_{CM} = \frac{M_x}{A} \tag{2.7}$$

$$y_{CM} = \frac{M_y}{A} \tag{2.8}$$

Essas integrais podem ser resolvidas por integrais de linha, que seguem a forma padrão mostrada na Equação 2.9.

$$\oint_C \vec{F} \cdot \mathbf{n} d\vec{\ell} \tag{2.9}$$

Estas integrais, por sua vez, podem ser resolvidas utilizando o Teorema de Green (Equação 2.10). Este teorema afirma que "sob condições adequadas, o fluxo exterior de um campo $\vec{F} = M\mathbf{i} + N\mathbf{j}$ através de uma curva fechada simples C é igual à integral dupla do divergente do campo sobre a região R limitada por C"(THOMAS et al., 2008).

$$\oint_C \vec{F} * n d\vec{\ell} = \oint_C (M dy + N dx) = \iint \left(\frac{\partial M}{\partial x} + \frac{\partial N}{\partial y} \right) dx dy \tag{2.10}$$

Dessa forma a principal dificuldade recai sobre a escolha de uma função $\vec{F}$ que se adeque ao esperado. Nos casos mostrados anteriormente é necessário encontrar uma função tal que $\left(\frac{\partial M}{\partial x} + \frac{\partial N}{\partial y} \right)$ seja igual a 1 para o cálculo da área, x para o momento em y, e y para o momento em x. Escolhendo os valores $\langle y/2, x/2 \rangle$ para a área, temos o resultado mostrado na Equação 2.11.

$$\int_{(P_0,P_1)} \vec{F} d\vec{\ell} = \frac{1}{2} (y_i x_{i-1} - y_{i-1} x_i) = W_1 \tag{2.11}$$

Aplicando esta técnica sucessivamente para os n pontos do *bounding box* e realizando as escolhas adequadas de $\vec{F}$, segue então que os valores para a área e momentos dos polígonos podem ser calculados de acordo com as fórmulas dadas nas equação 2.12, 2.13 e 2.14, respectivamente.

$$A = \sum_{i=1}^{n} W_i \tag{2.12}$$

$$M_x = \sum_{i=1}^{n} \frac{y_i + y_{i-1}}{3} W_i \tag{2.13}$$

$$M_y = \sum_{i=1}^{n} \frac{x_i + x_{i-1}}{3} W_i \tag{2.14}$$

Tais formas são mais vantajosas de serem utilizadas na codificação da solução proposta, visto que substituem integrais duplas por somatórios.

3 Desenvolvimento

Neste capítulo será descrito o desenvolvimento do trabalho. O capítulo é dividido em três seções: *Linguagem*, *Visualizador* e *API*. Na primeira parte são discutidas a sintaxe da linguagem, suas palavras reservadas, e como cada *feature* foi implementada. A segunda parte, *Visualizador*, mostra como este foi implementado, as ferramentas utilizadas e como ele se comunica com o interpretador. Finalmente, na seção *API*, discute-se a integração da linguagem proposta com a linguagem C.

3.1 Linguagem

Por se tratar de um trabalho de especificação de uma nova linguagem de domínio específico, é necessário primeiro estabelecer uma série de etapas de maneira a gerar especificação da mesma. Para este fim, adotou-se uma metodologia similar à utilizada por Alfred Aho, professor da disciplina de compiladores na Universidade de Columbia (BIANCUZZI; WARDEN, 2009). Esta metodologia consiste nos seguintes passos:

1. Decisão de escopo;
2. Escrita do *white paper* (SELTZER, 2014);
3. Especificação a linguagem:
    a) Descrição da linguagem em notação EBNF;
    b) Criação do manual da linguagem;
4. Implementação;
5. Testes de regressão.

Uma vez definida a estrutura da linguagem, deu-se início a etapa de implementação. Nesta fase foram utilizadas os *softwares Flex* e *GNU Bison*, ferramentas utilizadas para realizar análise léxica e sintática de códigos, respectivamente. Ambas as ferramentas são *open source*, utilizadas em conjunto com a linguagem C para criação de tradutores em geral.

Como mencionado, o *Flex* é responsável pela análise léxica, que consiste em atravessar o código e retornar os *tokens* específicos a cada lexema da linguagem através de chamadas à função `yylex()`, gerada automaticamente pelo *Flex* e que tem como base os lexemas definidos neste. Estes *tokens* são então utilizados pelo *Bison*, que realiza a análise

sintática do código e constrói a árvore de *parse*. Para isso a função `yyparse()` é invocada, que por sua vez chama *yylex()* para obter os *tokens* necessários para realizar a análise sintática. Estas informações são mostradas no diagrama da Figura 11.

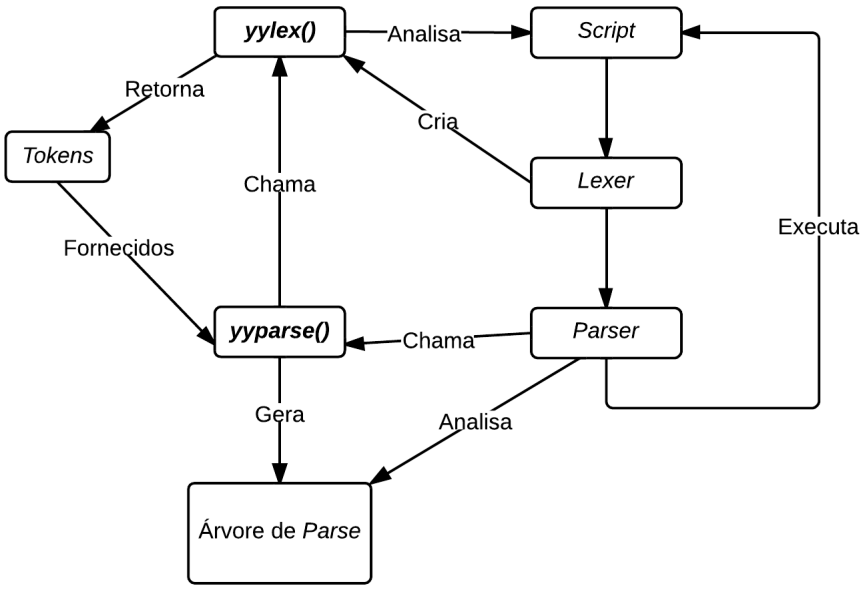

Figura 11 – Diagrama mostrando a relação entre as ferramentas.

Como o tratamento de colisões envolve cálculos no domínio dos números reais, a linguagem utiliza apenas dados do tipo *double* e não possui suporte para *chars* ou *strings*. A representação destes dados pode ser feita individualmente (escalares) ou em duplas (coordenadas). Coordenadas podem ser organizadas no formato de lista, que por sua vez pode ser utilizada para definir um *bounding box* genérico, sendo que formas básicas como quadrados e círculos são nativamente definidos e podem ser acessados através das palavras reservadas **`square`** e **`circle`**. O uso de escalares ou coordenadas depende do contexto.

Em sua interação com a linguagem C, Colli é usada de maneira iterativa, isto é, o usuário descreve um cenário onde ocorrerem as possíveis colisões a cada passo da simulação. Sendo assim, decisões e repetições podem ser feitas externamente, e então passadas para o interpretador. Dessa forma, o uso de estruturas de laços de repetição e testes condicionais é desnecessário entre os comandos da linguagem, fazendo com que ela não seja uma linguagem *Turing Complete*. Por fim foi decidido que o usuário não terá acesso direto às operações matemáticas básicas de soma, subtração, multiplicação e divisão, devendo utilizar combinações dos comandos disponíveis na linguagem para modificar os *bounding boxes*.

A Tabela 1 mostra as palavras reservadas da linguagem, assim como os *tokens* retornados pelo analisador sintático.

Expressão/Símbolo	**Classe**	***Token***
is	Palavras Reservadas	IS
bound by	Palavras Reservadas	BOUNDBY
scaled	Palavra Reservada	SCALED
translated	Palavra Reservada	TRANSLATED
rotated	Palavra Reservada	ROTATED
collisions with	Palavras Reservadas	COLLISIONTEST
square	Palavra Reservada	SQUARE
circle	Palavra Reservada	CIRCLE
reference	Palavra Reservada	REFERENCE
all	Palavra Reservada	all
show	Palavra Reservada	SHOW
scene	Palavra Reservada	SCENE
quit	Palavra Reservada	QUIT
{	Símbolo	BRA
}	Símbolo	KET
;	Símbolo	LISTSEPARATOR
,	Símbolo	VARSEPARATOR

Tabela 1 – Palavras reservadas, símbolos, e *tokens*.

Baseado nas informações levantas para a criação do *white paper* e nas *features* desejadas para a linguagem, foi decidido que os códigos desenvolvidos em Colli deveriam se aproximar da linguagem natural, buscando dar uma fluência e sequência lógica às expressões utilizadas. Esta decisão deve ficar clara nos elementos da linguagem, como por exemplo em expressões de definição e atribuição de valores a variáveis. Este conjunto sintático foi definido durante na etapa anterior do trabalho, utilizando a metodologia explicitada, sendo revisado e expandido com base no *feedback* recebido durante a apresentação do trabalho. As modificações serão comentadas ao longo desta seção.

3.1.1 Criação de *Bounding Boxes*

Antes de mostrar as diferentes maneiras para a criação de *bounding boxes*, é interessante que se saiba como um objeto é representado em memória. Os objetos são todos do tipo *doubleVector*, uma estrutura que contém os campos mostrados na Tabela 2.

O usuário dispõe de 3 maneiras distintas para a criação de *bounding boxes*:

1. Forma predefinida: O usuário pode utilizar as palavras reservadas *square* e *circle*, que criam um quadrado e um círculo, ambos tamanho 1 e com centro em $(0, 0)$;
2. Forma explicita: O usuário descreve o *bounding box* como uma lista de coordenadas;
3. Cópia: O usuário diz que o novo *bounding box* é idêntico a um *bounding box previamente declarado.*

Campo	Tipo	Significado
`self`	**`int`**	Índice do identificador na tabela de símbolos.
`vectorElems`	**`double*`**	*Array* contendo as coordenadas x e y dos vértices do objeto.
`size`	**`int`**	Número de elementos em `vectorElems`.
`capacity`	**`int`**	Capacidade máxima de `vectorElems`.
`type`	**`int`**	Variável indicando o tipo do objeto.
`centerX` e `centerY`	**`double`**	Coordenadas x e y do centro geométrico do objeto.

Tabela 2 – Campos para as `structs` de armazenamento.

O Código 3.1 mostra os três tipos de declaração, onde *bounding boxes* idênticos são criados das três maneiras mostradas.

Código 3.1 – Declação de objetos em Colli.

```
1 predefined is bound by square
2 explicit is bound by {0.5, -0.5; 0.5, 0.5; -0.5, 0.5; -0.5, -0.5}
3 copy is bound by predefined
```

3.1.2 Informações sobre um objeto

Uma das primeiras mudanças realizadas para a segunda etapa de trabalho foi a adição de um comando que mostrasse em tela informações sobre um *bounding box* existente. Para isso foi adicionado à EBNF a sequência `show` *identificador*. Este comando mostra informações sobre o objeto desejado, como as coordenadas do centro geométrico do mesmo, bem como as coordenadas de seus vértices. O Código 3.2 exemplifica o funcionamento desta *feature*.

Código 3.2 – Obtendo informações sobre um objeto.

```
1 object is bound by square scaled 20.00 translated 20.0, 20.0
2 show object
```

Após a execução deste código, as seguintes informações seriam mostradas na tela:

```
Identifier: object
Center: 20.00, 20.00
Size: 4
Contents:
        { 30.00, 30.00 ; 10.00, 30.00 ; 10.00, 10.00 ; 30.00, 10.00 }
```

as quais são, em ordem, a localização do centro geométrico, o número de pares de coordenadas que constituem o objeto, e as coordenadas dos vértices, separadas por `';'`.

3.1.3 Verificação de erros de sintaxe

Durante a etapa de *parse* do código, caso o *Bison* detecte algum erro de sintaxe, uma chamada para a função `yyerror()` é realizada (LEVINE, 2009). Por padrão, esta função retorna a apenas mensagem *"syntax error"*, porém é possível redefini-la de maneira a produzir mensagens de erro mais clara, indicando exatamente o *token* que causou o erro. Além de ser chamada automaticamente em casos de erro de sintaxe, é possível chamar a função manualmente, de acordo com algum teste realizado.

Por se tratar de uma linguagem interpretada, é interessante que o intepretador seja capaz de se recuperar em casos de erro. Isto pode ser realizado definindo-se um *token* `error`. Este *token* é declarado implicitamente, não necessitando ser declarado no *parser*. Ao detectar algum erro, o *token* é enviado ao *parser*, sendo que a maneira como o interpretador trata este erro é realizada da mesma forma que as outras regras de *parse* são realizadas. Para que o interpretador seja capaz de se recuperar do erro, é necessário que sejam utilizadas duas macros: `yyerrok` e `yyclearin`. A primeira é sinaliza para o *parser* continuar normalmente, e a segunda descarta os *tokens* de *lookahead*, se houverem.

No Código 3.3 é mostrada a maneira como a função `yyerror()` foi redefinida para que indicasse algum identificador desconhecido.

Código 3.3 – Redefinição da função `yyerror()`.

```
1  int yyerror(char *s, char *id) {
2
3          extern char* yytext;
4          extern int yylineno;
5
6          if(!id)
7                  fprintf(stderr, "Error at (%d) \"%s\". %s\n"
8                          , yylineno, yytext, s);
9          else
10                 fprintf(stderr, "Error at (%d). Unknown identifier \"%s\"\n."
11                                 , yylineno, id);
12
13         return 0;
14 }
```

3.2 Visualizador

Como uma ferramenta de auxílio ao desenvolvedor, foi criado um *script* de visualização, desenvolvido utilizando a versão 2.7.6 da linguagem *Python*[1] juntamente com a biblioteca *PyGame*, um conjunto de módulos *Python* desenvolvidos sobre a biblioteca SDL. Esta escolha de ferramentas se deve ao suporte nativo da linguagem *Python* possui a estruturas de mais alto nível, como tuplas e listas, agilizando o desenvolvimento.

O visualizador cria uma tela de resolução 800x600 *pixels* onde todos os objetos criados até o momento são desenhados. Esta tela utiliza o sentido padrão de computação gráfica para a orientação dos eixos: eixo x positivo à direita, eixo y positivo para baixo, com a origem do sistema de coordenadas localizada no canto superior esquerdo da tela. Esta representação é como uma fotografia dos *bounding boxes* em determinado instante, tornando possível ao usuário analisar com calma a disposição espacial dos objetos, verificando por inspeção os casos de colisão.

Basicamente, o programa realiza 3 etapas para desenhar o objeto na tela: inicialização, leitura e escolha da cor. Durante a inicialização é criada uma referência para a tela, com as propriedades listadas anteriormente. Em seguida, o programa recebe uma série de entradas em um formato específico, utilizando a função `input()` do *Python*. Esta função busca casar a entrada recebida com algum padrão de estrutra de dados do *Python*. No caso, utiliza-se o formato de valores entre parênteses e separados por vírgula (i.e., (1.0, 2.0)), que são lidos como tuplas. A escolha da cor ocorre antes do objeto ser desenhado na tela, sendo realizada pela função mostrada no Código 3.4:

Código 3.4 – Função para escolha aleatória de cor.

```
1  import pygame
2  from pygame.locals import *
3  from math import *
4
5  import itertools as it
6  import random as rnd
7
8  #Randomly generates a color
9  def color():
10         levels = [0, 128, 64, 255]
11         c = it.product(levels, repeat = 3)
12         res = []
13
14         for i in c:
```

[1] https://www.python.org/

```
15                res.append(i)
16
17        return rnd.sample(res, 50)
18
19 def colorPicker():
20        colorPicker.counter += 1
21
22        if colorPicker.colors[colorPicker.counter] == (255, 255, 255):
23                colorPicker.counter+=1
24
25        return colorPicker.colors[colorPicker.counter % 50]
```

A função `color` gera uma lista de tuplas de tamanho 3, contendo amostras sorteadas entre os valores definidos em `levels`. Os valores gerados são lidos sequencialmente, pulando a cor branca para que o objeto não se misture ao fundo.

O padrão de leitura consiste em uma linha com o número de objetos a serem desenhados, seguidos pelas coordenadas que definem o polígono, no formato de tuplas. Um caso específico é o dos círculos, que necessitam da localização do centro e do tamanho do raio. Isso é passado ao visualizador como uma tupla com as coordenadas `centerX` e `centerY` do objeto e outra tupla redundante com o raio em ambas posições.

Finalmente, os objetos passados são então desenhados na tela. Para requisitar ao interpretador que a cena de colisão seja mostrada deve-se utilizar o comando **`show scene`**. No Código 3.5 é mostrado como invocar o visualizador na sintaxe Colli, e o resultado exibido pelo visualizador é mostrado na Figura 12.

Código 3.5 – Invocação do visualizador

```
1 first is bound by circle scaled 150.0 translated 470.0, 500.0
2 second is bound by square scaled 50.0 rotated 45.0 translated 50.0, 50.0
3 third is bound by {0.0, 1.0; 3.0, 4.5; 1.2, 3.7} scaled 100.0 translated
    400.0, 300.0
4
5 fourth is bound by second scaled 0.5 rotated -23.7 translated 30.0, -30.0
6 fifth is bound by third rotated 90.0 translated -50.0, 0.0
7 sixth is bound by circle scaled 30.0 translated 800.0, 600.0
8
9 show scene
```

Internamente, a comunicação entre os dois módulos funciona da seguinte maneira: o interpretador cria um arquivo chamado *description.scene*, contendo as informações no formato descrito anteriormente, em seguida o processo realiza um `fork()`, e o processo

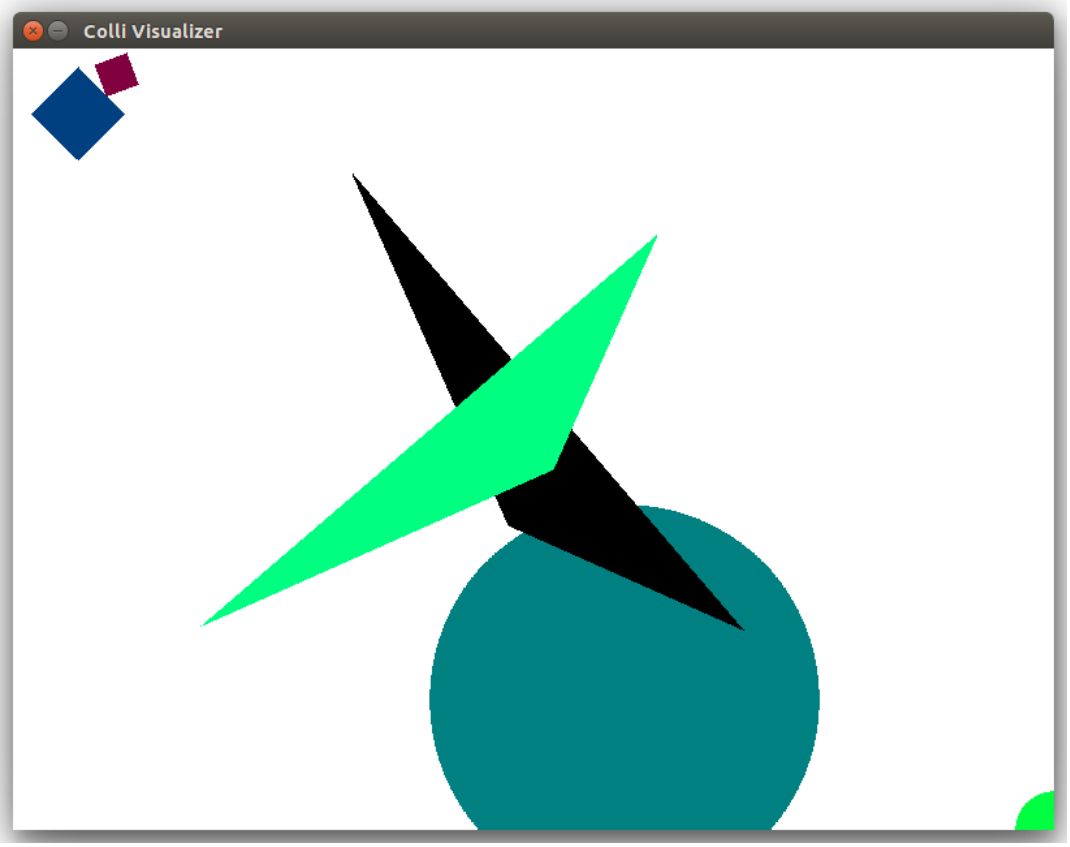

Figura 12 – Visualização da cena de colisão descrita no código.

filho executa um *bash script* que simplesmente chama o interpretador *Python* para executar o arquivo do visualizador, e redireciona para sua entrada o arquivo *description.colli* (isto é, o *script* executa o comando *python visualizer.py* < *description.colli*).

No caso mostrado, o arquivo gerado tem o formato mostrado a seguir.

```
6
(470.0000, 500.0000), (150.0000, 150.0000)
(85.3553, 50.0000),(50.0000, 85.3553),(14.6447, 50.0000),(50.0000, 14.6447)
(254.8000, 78.8000),(554.8000, 428.8000),(374.8000, 348.8000)
(96.1868, 27.1055),(72.8945, 36.1868),(63.8132, 12.8945),(87.1055, 3.8132)
(127.0222, 449.9111),(477.0222, 149.9111),(397.0222, 329.9111)
(800.0000, 600.0000), (30.0000, 30.0000)
```

3.3 API

Um dos principais objetivos do desenvolvimento deste projeto consiste na criação da API de integração da solução proposta com a linguagem C. Para o desenvolvimento de tal solução, escolheu-se como base a linguagem LUA e como a mesma inte-

rage com a linguagem C. Neste contexto, a comunicação é feita através de uma pilha (IERUSALIMSCHY, 2014), que deve ser criada pelo programa em C. Através das funções disponibilizadas por uma API, o usuário deve colocar na pilha as funções LUA que deseja executar, bem como os parâmetros requisitados pela função.

O uso da estratégia da pilha se mostra interessante pois facilita a comunicação entre uma linguagem estaticamente tipada e uma dinâmicamente tipada (IERUSALIMSCHY, 2014), como é o caso da linguagem Colli. No caso da API para Colli, utiliza-se uma pilha que recebe valores do tipo `string` e do tipo `int`, representando informações necessárias para executar um comando, e o tipo de cada item na pilha. A pilha também contém um campo que guarda o número de itens empilhados. Os tipos, definidos por um valor da enumeração `StackDefs`, são:

Comando: Na pilha, este valores representam os comandos da linguagem, sendo colocados na mesma através da função `colliPushCommand`, que recebe como parâmetro um inteiro. Para aumentar a clareza do código, os valores possíveis foram declarados como parte da enumeração `StackDefs`.

Identificador: Representa o nome dado a um dos objetos. Estes valores são colocados na pilha diretamente na forma de `string`.

Valor: Representa os valores para rotação, contração/expansão linear e translação. Estes valores devem ser enviados como `double`, para serem colocados na pilha.

Forma: Estes valores representam os possíveis formatos dos *bounding boxes*. Os valores passados devem ser `strings` como `"circle"`, `"square"` ou uma lista no formato esperado pelo interpretador Colli (isto é, `"{0.0, 0.0; 1.0, 1.0; 1.0, 0.0}"`).

O Código 3.6 mostra como a pilha foi declarada no código.

Código 3.6 – Pilha utilizada na API.

```
1 typedef struct ColliStack_{
2
3         int command[STACKSIZE];
4         char *parameters[STACKSIZE];
5         int type[STACKSIZE];
6         int size;
7
8 } ColliStack;
```

A pilha deve ser inicializada utilizando a função `colliInit()`, que retorna um ponteiro para uma pilha de tamanho 20. Como o espaço utilizado pela pilha é liberado

novamente após o comando ter sido executado, uma pilha com apenas 20 posições é suficiente para alcançar a funcionalidade proposta. Os parâmetros devem ser empilhados utilizando as funções apropriadas para o que se deseja empilhar. Por exemplo, para empilhar um identificador para um objeto a ser criado, a função `colliPushId()` deve ser chamada, sendo o mesmo princípio utilizado para empilhar comandos, valores e formas. As funções `colliPushId()`, `colliPushShape()` colocam na pilha exatamente o parâmetro passado a elas, já a função `colliPushCommand()` realiza um *switch* com o parâmetro recebido, colocando a *string* apropriada na pilha. Por sua vez, a função `colliPushValue()` realiza uma conversão de `double` para `string` e coloca este valor na pilha.

Uma vez empilhados os valores, o usuário deve chamar a função `colliRun()`. Esta função verifica se o parâmetro no topo da pilha é um comando, buscando os parâmetros apropriados na pilha para executar automaticamente. Entre os comandos Colli, apenas as requisições de teste de colisão fogem deste padrão, sendo realizadas utilizando a função `colliTest()`, que recebe como parâmetro o identificador sob o qual deseja-se realizar o teste de colisão, e retorna um vetor de `strings` que contém os identificadores dos objetos com os quais houve colisão.

De forma geral, os valores não necessitam ser empilhados em uma ordem específica, exceto quando deseja-se executar uma operação de translação, na qual deve-se colocar na pilha primeiro o valor do *offset* em x, e em seguida o valor do *offset* em y. O Código 3.7 mostra como proceder para criar dois objetos e realizar um teste de colisão entre os mesmos.

Código 3.7 – Utilização de Colli na linguagem C.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include "colli.h"
5
6  int main(){
7
8          int i = 0;
9          char **results = NULL;
10         int totalCollisions = 0;
11
12         ColliStack *s = colliInit();
13
14         colliPushId(s, "first");
15         colliPushShape(s, "circle");
16         colliPushCommand(s, BIND);
```

```
17          colliRun(s);
18
19          colliPushId(s, "first");
20          colliPushCommand(s, SHOW);
21          colliRun(s);
22
23          colliPushId(s, "second");
24          colliPushShape(s, "circle");
25          colliPushCommand(s, BIND);
26          colliRun(s);
27
28          colliPushId(s, "second");
29          colliPushValue(s, -0.5);
30          colliPushValue(s, 0.5);
31          colliPushCommand(s, TRANSLATE);
32          colliRun(s);
33
34          colliPushId(s, "second");
35          colliPushCommand(s, SHOW);
36          colliRun(s);
37
38          results = colliTest("first", &totalCollisions);
39
40          if(results){
41                  for(i = 0; i < totalCollisions; i++)
42                          printf("first collided with %s\n", results[i]);
43          }
44
45          return 0;
46  }
```

A função `colliTest()` irá retornar um *array* de `strings` contendo os identificadores dos objetos que estão colidindo com o objeto passado como parâmetro. No Código 3.8 são mostrados os comandos equivalentes em sintaxe Colli para executar o teste do Código 3.7.

Código 3.8 – Código Colli equivalente.

```
1 first is bound by circle
2
3 second is bound by circle
```

```
4 second is translated -0.5, 0.5
5
6 collisions with first
```

Em ambos os casos, o resultado mostrado na tela deve ser o texto `first collided with second`.

A comunicação entre o interpretador e as funções da API se dá da seguinte forma: ao ser chamada, a função de inicialização cria uma nova *thread* que executa o interpretador em *background*. Tanto a API quanto o interpretador abrem um *pipe* para comunicação, sendo que a API abre o mesmo somente para escrita, e o interpretador somente para leitura.

A API desenvolvida utiliza semáforos para sincronização entre a *thread* do interpretador, e a *thread* principal, que envia as mensagens a serem interpretadas. Estes semáforos são criados na função `colliInit()` (mostrada no Código 3.9), e são usados para sinalizar que uma mensagem pode ser enviada para o interpretador (`canSend`), e que o interpretador pode ler uma mensagem (`canRead`). Estes semáforos são utilizados de maneira alternada em cada parte do código: a API realiza um *wait* em `canSend`, verificando se a mensagem pode ser enviada, e um *post* em `canRead` após o envio da mensagem. O interpretador realiza um *wait* em `canRead`, e após executar o comando recebido, um *post* em *canSend*, sinalizando que está pronto para receber a próxima mensagem.

Código 3.9 – Inicialização da API Colli.

```
1 ColliStack *colliInit(){
2
3       ColliStack *stack;
4
5       char *sink = "/tmp/colliPipe";
6       mkfifo(sink, 0666);
7
8       canRead = sem_open("/canRead", O_CREAT, 0644, 0);
9       canSend = sem_open("/canSend", O_CREAT, 0644, 1);
10
11      if(canRead == SEM_FAILED || canSend == SEM_FAILED)
12              printf("API: %s\n", strerror(errno));
13
14      stack = (ColliStack *)malloc(sizeof(ColliStack));
15      stack->size = 0;
16
17      pthread_create( &interpreterThread, NULL, interpreter, NULL);
18
```

```
19          return stack;
20  }
```

Inicialmente o uso destes semáforos não era para suficiente garantir que o interpretador estaria pronto para receber o primeiro comando, o que se devia à maneira como a função `yyparse()` se comporta. O primeiro *post* em `canSend` é realizado para sinalizar à API que esta já pode enviar mensagens para o interpretador, porém a função `yyparse()` é blocante, não sendo possível realizar o *post* ao semáforo após sua chamada. Utilizando uma aproximação de uma única *thread* no interpretador, restam duas opções: realizar o *post* dentro da função de *parse*, ou imediatamente antes da mesma ser chamada.

A primeira opção seria a ideal, já que seria possível postergar a execução do *parse* até que uma mensagem tivesse sido realmente enviada. Entretanto, isto não pode ser realizado devido ao fato de que a função `yyparse()` faz parte da API de integração do *Bison* com a linguagem C, não sendo possível reescrever esta função de maneira que ela se adeque ao necessário. Relizando o *post* antes da chamada, é possível que o sistema operacional decida que o *time-slice* da *thread* acabe imediatamente depois do *post* e antes da chamada para a função `yyparse()`, o que também leva a problemas de sincronia.

Após algumas tentativas com diferentes configurações para a ordem das operações com semáforos, decidiu-se realizar uma pequena modificação na função *main()* do interpretador. Tal modificação consistiu na criação de uma *thread* que realiza a chamada da função `yyparse()`, sendo possível então realizar o *post* sinalizando que o interpretador está pronto para realizar o *parse* dos comandos enviados após a inicialização desta *thread*.

4 Resultados e Discussão

Neste capítulo são discutidos os resultados obtidos em cada aspecto do trabalho, apontando os pontos nos quais o trabalho foi bem sucedido, e também os pontos nos quais há a possibilidade de melhoria do mesmo.

4.1 Interpretador Colli

Todo conjunto sintático proposto foi implementado: a criação e manipulação de *bounding boxes* ocorre de acordo com o previsto, como mostrado na Figura 13. Entretanto, em termos de implementação, o código necessita de refatoração, especialmente no que diz respeito à eliminação de *memory leaks*, o que se deve à quantidade de informação alocada para a criação e execução da árvore de *parse* e que não é liberada ao fim da execução.

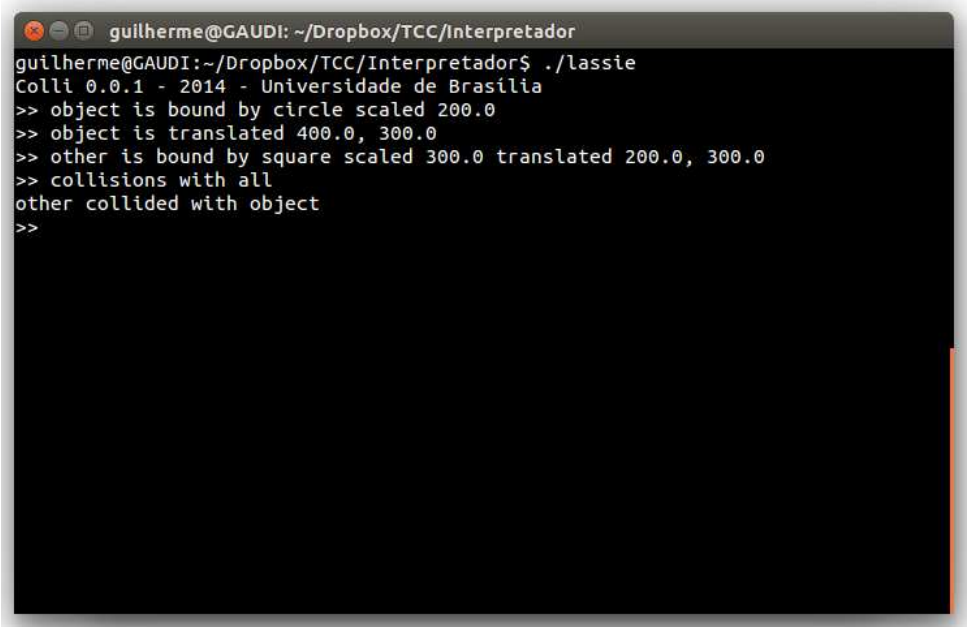

Figura 13 – Interpretador em funcionamento.

Além disso o interpretador também precisa de melhorias no seu sistema de *error report*, de forma a gerar informações mais precisas acerca da causa do erro de execução, bem como utilizar um padrão para tais mensagens. Na versão atual é por vezes difícil identificar a causa do erro (sintaxe, alocação de memória, além de alguns erros não tratados que derrubam o interpretador). Apesar destes problemas, o interpretador se comporta de forma adequada quando os comandos são inseridos corretamente.

Em relação aos testes de detecção de colisão, faz-se necessário uma análise do custo computacional dos mesmos, de maneira a determinar os "gargalos" das rotinas implemen-

tadas e, se possível, diminuir seu impacto. Os testes foram implementados somente para polígonos convexos, sendo também interessante implementar uma forma de testes para polígonos côncavos.

4.2 Visualizador

O visualizador foi implementado em sua totalidade, realizando de maneira satisfatória o seu objetivo. Neste caso também seria interessante uma análise de custo computacional com ferramenta de *profiling*, já que a maneira que o sistema é executado agora (através de uma chamada de sistema que executa um *script*) pode acarretar atrasos de execução. O processo de execução pode ser agilizado realizando um *port* do código *Python* para C, utilizando a biblioteca SDL. A Figura 14 mostra a cena descrita com os comandos da Figura 13

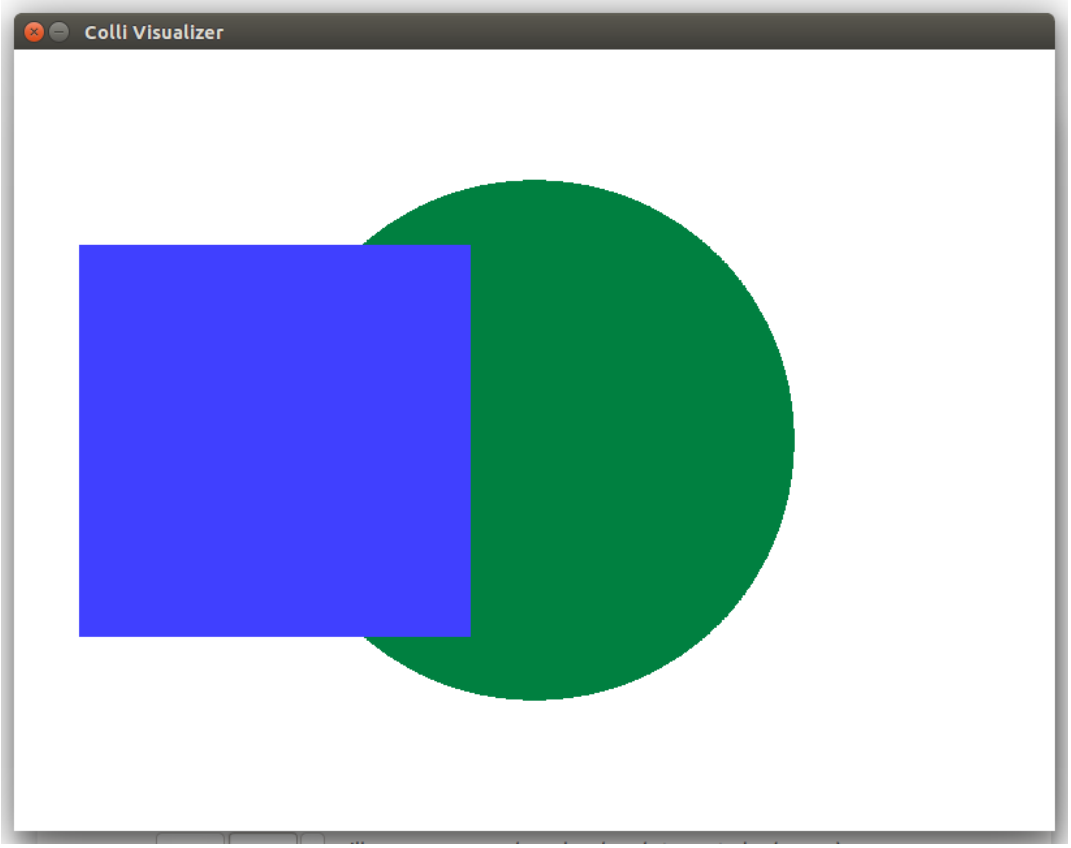

Figura 14 – Visualizador em funcionamento.

4.3 API

A API de integração foi implementada de maneira satisfatória, sendo possível realizar a criação e teste de colisão de *bounding boxes* na linguagem C, utilizando as funções desenvolvidas. Entretanto, a solução ainda mostra alguns *bugs*, como por exemplo

a perda de sincronização entre os módulos durante a execução. Alguns comandos Colli também não foram implementados na API, não podendo ser utilizados desta forma.

Também neste caso seria interessante o uso de uma ferramente de *profiling* para avaliar a existência de *memory leaks* e custo computacional. O uso de *strings* como resultado do teste também pode se mostrar uma razão de atraso durante a execução, sendo necessária uma mudança na forma como o resultado do teste é comunicado de volta à API. Entretanto, tal mudança deve acarretar alterações a nível da implementação do interpretador, relativas à maneira como o teste é realizado.

Função	**Parâmetros**	**Descrição**	**Retorno**
`colliInit()`	`void`	Inicializa a API.	`colliStack *`
`colliPushCommand()`	`colliStack*, int`	Empilha um comando.	`int`
`colliPushId()`	`colliStack*, char*`	Empilha um identificador.	`int`
`colliPushShape()`	`colliStack*, char*`	Empilha um formato de *bounding box* (`circle`, `square` ou uma lista).	`int`
`colliPushValue()`	`colliStack*, double`	Empilha um valor.	`int`
`colliRun()`	`colliStack*`	Executa o comando empilhado.	`int`
`colliTest()`	`char*, int*`	Requisita um teste de colisão sobre um identificador.	`char**`
`colliClose()`	`colliStack*`	Encerra o uso da API.	`int`

Tabela 3 – Funções da API Colli.

4.4 Comentários Gerais

A sintaxe desenvolvida para a linguagem Colli até o momento buscou criar apenas um conjunto de palavras que pudesse ser utilizado na criação e realização testes de colisão entre objetos simples, não proporcionando nenhuma maneira para informar como o teste deve ser feito, ou dando suporte a outros tipos de representação de *bounding boxes*. Uma possibilidade interessante de expansão para a sintaxe seria o suporte a criação de "camadas" no ambiente de colisão e criação de hierarquia de testes, buscando um aumentar a velocidade dos testes.

4.5 Trabalhos Futuros

Em seu atual estado, Colli desempenha seu propósito de maneira satisfatória, porém não eficiente. Trabalhos futuros devem envolver a questão de otimização de seu funcionamento, de forma a permitir que a linguagem seja utilizada no desenvolvimento de jogos. Como mencionado na Seção 4.1, expandir o conjunto sintática para dar suporte à funções mais complexas também agregaria valor na linguagem enquanto solução genérica para o desenvolvedor.

5 Conclusão

O trabalho elucidado ao longo deste documento teve três principais objetivos: desenvolver uma nova linguagem de programação com foco em colisões 2D, implementar uma ferramenta de visualização para as cenas descritas com a linguagem proposta, e desenvolver uma API de integração da lingaugem criada com a linguagem C. Estes objetivos foram alcançados de maneira satisfatória, gerando um conjunto de ferramentas funcionais que podem ser utilizadas no desenvolvimento de jogos simples, como planejado inicialmente.

Entretanto, as ferramentas se encontram em um estado inicial em termos de estabilidade, não podendo ser apresentadas ao público como um produto. Para tal, faz-se necessário uma análise completa do código desenvolvido, buscando otimizar e resolver os *bugs* encontrados no mesmo, bem como uma melhora na usabilidade do interpretador.

Referências

AHO, A. et al. *Compiladores*: Princípios, técnicas e ferramentas. 2. ed. São Paulo, Brasil: Pearson, 2013. Citado 3 vezes nas páginas 23, 24 e 25.

ANDREW, A. Another efficient algorithm for convex hulls in two dimensions. In: *Information Processing Letters*. [s.n.], 1979. v. 9, n. 5, p. 216–219. Disponível em: <http://dx.doi.org/10.1016/0020-0190(79)90072-3>. Citado na página 28.

BARNEY, B. *POSIX Threads Programming*. 2014. Disponível em: <https://computing.llnl.gov/tutorials/pthreads/#Pthread>. Citado na página 37.

BIANCUZZI, F.; WARDEN, S. *Masterminds of Programming: Conversations with the Creators of Major Programming Languages*. California, EUA: O'Reilly Media Inc., 2009. 111-113 p. Citado na página 41.

BITTLE, W. *Code Zealot: SAT (Separating Axis Test)*. 2014. Disponível em: <http://www.codezealot.org/archives/55>. Citado 3 vezes nas páginas 15, 30 e 31.

BOLDRINI, J. L. et al. *Álgebra Linear*. Campinas, Brasil: Editora HARBRA, 1980. 147-150 p. Citado na página 38.

ERICSON, C. *Real Time Collision Detection*. Florida, EUA: CRC Press, 2004. Citado 6 vezes nas páginas 15, 26, 27, 28, 29 e 34.

GOLDT, S. et al. *Linux Interprocess Communications*. 2014. Disponível em: <http://tldp.org/LDP/lpg/node7.html#SECTION00700000000000000000>. Citado na página 37.

IERUSALIMSCHY, R. *Programming in Lua*. 2014. Disponível em: <http://www.lua.org/pil/contents.html>. Citado na página 49.

LEVINE, J. *Flex & Bison*. California, EUA: O'Reilly Media Inc., 2009. Citado na página 45.

SELTZER, M. A. *Writing White Papers*. 2014. Disponível em: <http://www.writingwhitepapers.com/resources.html>. Citado na página 41.

STRANG, G. *Introduction to Linear Algebra*. Massachusetts, EUA: Wellesley–Cambridge Press, 2009. Citado na página 37.

TANENBAUM, A. S. *Sistemas Operacionais Modernos*. 3. ed. São Paulo, Brasil: Pearson, 2010. Citado na página 36.

THOMAS, G. B. et al. *Cálculo*. São Paulo, Brasil: Addison Wesley, 2008. Citado 2 vezes nas páginas 38 e 39.

W3C. *XQuery 1.0: An XML Query Language*. 2014. Disponível em: <http://www.w3.org/TR/2010/REC-xquery-20101214/#id-grammar>. Citado na página 24.

WESTHOFF, J. *Calculate a Convex Hull - The Quickhull Algorithm*. 2014. Disponível em: <http://www.westhoffswelt.de/blog/2009/10/21/calculate-a-convex-hull-the-quickhull-algorithm>. Citado na página 28.

Anexos

ANEXO A – EBNF Colli

```
/*Basic Rules*/

digit ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
character ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" |
     "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v"
     | "w" | "x" | "y" | "z" | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "
    H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" |
     "T" | "U" | "V" | "W" | "X" | "Y" | "Z"


number ::= ("+"|"-")?(digit)+"."(digit)+
identifier ::= character(character|number)*

/*Extended Rules*/

coordinate ::= number "," number
idListElems ::= (identifier";")* identifier
coordlist ::= "{" coordListElems "}"
coordListElems ::= (coordinate";")* coordinate

/*Advanced Rules*/

script ::= statement|(statement "\n" script)
statement ::= (assignmentstmt|collisionstmt|referenceshift|visualisestmt)
    ?
assignmentstmt ::= identifier "is" (boxstmt|scalestmt|translationstmt|
    rotationstmt|coordList|quitstmt|printObj|visualisestmt)

boxstmt ::= "bound by" (identifier|coordlist|"square"|"circle") optional
optional ::= (options?options?options?)
options ::= translationstmt|rotationstmt|scalestmt
scalestmt ::= "scaled" number
translationstmt ::= "translated" coordinate
rotationstmt ::= "rotated" number
collisionstmt ::= "collisions with" ("all"|identifier)
```

```
30 referenceshift ::= "reference is" coordinate
31 quitstmt = "quit"
32
33 printObj ::= "show" identifier
34 visualisestmt ::= "show scene"
```

ANEXO B – Manual Colli

Colli é uma linguagem para aqueles que querem agilidade no processo de desenvolvimento de suas simulações físicas, especialmente no caso do desenvolvimento de jogos. Colli busca atingir este objetivo através da sua sintaxe reduzida. As palavras reservadas da linguagem são mostradas na Tabela 4.

Palavras Reservadas
`is`
`bound by`
`scaled`
`translated`
`rotated`
`collisions`
`with`
`square`
`circle`
`reference`
`all`
`show`
`scene`
`quit`

Tabela 4 – Palavras reservadas da linguagem Colli.

A linguagem foi projetada para que, utilizando apenas estas palavras reservadas, o usuário fosse capaz de descrever diferentes tipos de cenários onde podem ocorrer colisões. Para isso é possível especificar ponto a ponto a geometria de colisão, utilizando listas de pontos para descrever cada objeto, sendo possóvel modificar estes objetos ao longo da execução utilizando os comandos `translated`, `rotated` e `scaled`.

Nas seções seguintes o usuário será apresentado aos básicos da linguagem, iniciando com um teste simples de colisão entre *bounding boxes* circulares e progredindo para exemplos mais complexos nas seções seguintes.

B.1 *Hello Colli*

Na veia do tradicional *Hello world* em outras linguagens de programação, o primeiro programa em Colli é um exemplo mínimo que gera um *output*. O Código B.1 mostra o *script* do “Hello World” em Colli.

Código B.1 – Código básico Colli.

```
1 firstCircle is bound by circle
2 secondObject is bound by circle
3 collisions with all
```

Neste código, objetos são criados utilizando a expressão *identificador* **is** *forma*. Isso cria uma variável de nome *identificador*, que representa a forma especificada no parâmetro mais a direita da expressão. Colli conta com duas formas pré-definidas: **square** e **circle**. Utilizar esta expressão sem mais nenhum parâmetro irá gerar uma forma de lado ou raio *1.0*, e com centro na origem do sistema de coordenadas[1]. Ao se especificar uma lista de coordenadas, o centro geométrico do objeto descrito é calculado a partir das coordenadas passadas.

No exemplo, foram criados dois círculos de raio *1.0*, ambos centrados na origem do sistema de coordenadas. Obviamente estas formas geométricas estão colidindo, portanto o teste de colisão deve indicar que os dois objetos colidem. Para requisitar a realização do teste de colisão, deve-se utilizar o comando **collisions with** *objetos*. O usuário pode especificar sobre quais objetos deseja realizar o teste, mas inicialmente será utilizado o especificador **all**, que faz com que sejam realizados testes sobre todos os objetos.

Para executar um *script* em Colli, o usuário deve invocar seu interpretador, denominado *Lassie*. Se chamado sem parâmetros, o interpretador irá aguardar comandos do usuário, executando-os um por vez, à medida que são inseridos. Uma vez executado, o interpretador irá gerar a seguinte saída no terminal:

```
firstCircle collided with secondCircle
secondCircle collided with firstCircle
```

B.2 Obtendo informações dos objetos criados

Colli conta também com um comando que pode ser utilizado pelo usuário para verificar informações acerca do objeto criado. Para isso, o usuário utiliza a sintaxe **show** *identificador*. Isso irá gerar uma resposta da seguinte forma:

```
Identifier: <identificador>
Center: <coordenadas do centro>
Type: <tipo do objeto>
Size: <número de vértices do objeto>
Contents: <coordenadas dos vértices>
```

[1] O sistema de referência usado em Colli segue o padrão para aplicações em Computação, no qual o sentido positivo do eixo y é o oposto do usualmente adotado.

Para obter informações acerca dos *bounding boxes* do exemplo da seção anterior, basta adicionar as linhas apropriadas ao final do código, ficando da forma mostrada no Código B.2.

Código B.2 – Código para visualizar as informações dos objetos.

```
1 firstCircle is bound by circle
2 secondObject is bound by circle
3 collisions with all
4
5 show firstCircle
6 show secondCircle
```

Este código produz o seguinte *output*:

```
Identifier: firstCircle
Center: 0.00, 0.00
Type: Circle Shape
Size: 1
Contents: {1.00, 1.00}
Identifier: secondCircle
Center: 0.00, 0.00
Type: Circle Shape
Size: 1
Contents: {1.00, 1.00}
```

Vale também notar que em Colli, os círculos são representados por um único par de coordenadas de coordenadas redundante, que correspondem ao raio do círculo.

B.3 Reposicionando *bounding boxes*

O exemplo mostrado anteriormente demostra a sintaxe básica da linguagem, porém não possui aplicação prática. Naturalmente, se faz necessária a possibilidade de posicionar *bounding boxes* de acordo com a vontade do usuário. A primeira forma para realizar isto é mudar o referencial do sistemas de coordenadas. Isso pode ser feito associando um novo par de coordenadas à palavra reservada **`reference`**.

```
1 firstCircle is bound by circle
2 reference shift 10.00, 10.00
3 secondCircle is bound by circle
4 collisions with all
```

O código mostrado cria um círculo centrado em (0.0, 0.0), modifica o ponto de referência do sistema para (10.00, 10.00), e cria um novo círculom que agora centrado no ponto de referência definido anteriormente. Ao requisitar um teste de colisão sobre os objetos, nada é mostrado na tela, pois nenhum objeto está colidindo. É importante notar que os objetos definidos anteriormente não mudam de posição, mas possuem suas coordenadas alteradas. Objetos criados após a troca de referencial se baseiam no valor do mesmo para posicionar-se.

Outra maneira de realizar a translação de um objeto é utilizando o comando **translated**. Este deve ser seguido por um par de coordenadas que representam o centro do objeto sendo transladado. Este comando modifica apenas a posição do objeto desejado, não alterando as coordenadas de nenhum outro. Este comando utiliza coordenadas relativas, isto é, um objeto centrado em $(20.0, 20.0)$ e transladado em $(50.0, -10.0)$ tem a posição de seu centro alterada para $(70.0, 10.0)$.

```
1 firstCircle is bound by circle
2 secondCircle is bound by circle translated 10.00, 10.00
3 collsions with all
4 firstCircle is translated 9.50, 10.00
5 collisions with all
```

Este código cria novamente dois círculos, porém o segundo é transladado no momento de sua criação. Na primeira vez que um teste de colisão é requisitado, o sistema não indicará colisão. Em seguida, *firstCircle* tem seu centro transladado para a posição 9.5, 10.0, e ao requisitar novamente um teste de colisão, o seguinte resultado deve ser indicado no console:

```
firstCircle collided with secondCircle
secondCircle collided with firstCircle
```

B.4 *Bounding boxes* arbitrários

Como mencionado, além de **circle**, Colli também suporta a declaração de quadrados como formas de *bounding box* suportadas nativamente. Entretanto, é desejável que seja possível especificar formas geométricas convexas arbitrárias para os *bounding boxes*. Para isso o usuário pode definir uma lista de coordenadas que descreve um polígono qualquer. As listas em Colli devem estar entre chaves e utilizam o símbolo "*;*" como separador de elementos.

Código B.3 – Criação de *bounding box* polígonal.

```
1 firstObject is bound by {0.0, 0.0; 1.0, 0.0; 1.0, 1.0}
2 secondObject is bound by square translated 1.0, 0.5
```

```
3
4 collisions with firstObject
```

No *script* mostrado no Código B.3, primeiramente é criado um *bounding box* arbitrário no formato de um triângulo. Através do uso de listas de coordenadas o usuário pode criar *bounding boxes* da forma que desejar, porém deve-se tomar cuidado para que os pontos especificados não constituam um polígono côncavo[2]. Na segunda linha é criado um *bounding box* quadrado de lado 1.

Dessa forma, o usuário deve realizar ajustes nos *bounding boxes* criados, podendo utilizar os comandos **scaled**, **rotated** e **translated**. Os dois primeiros comandos recebem como parâmetro um número qualquer, e assim como o comando **translated** podem ser utilizados no momento da criação do objeto ou posteriormente através do padrão de associação de valores (*identificador* **is** *comando*).

O valor passado ao comando **rotated** é o ângulo em graus para rotacionar o *bounding box*, caso o valor seja positivo o *bounding box* é rotacionado no sentido anti-horário, caso seja negativo o objeto é rotacionado no sentido horário. Esta rotação é realizada em torno do centro da forma geométrica, e não da origem.

Código B.4 – Uso dos comandos *rotated* e *scaled*.

```
1 reference is 400.0, 300.0
2
3 firstObject is bound by {0.0, 0.0; 1.0, 0.0; 1.0, 1.0} scaled 200.00
4 secondObject is bound by square scaled 100.0 rotated 45.0
5
6 collisions with firstObject
7
8 show scene
```

No Código B.4, os mesmos objetos descritos anteriormente são criados, porém, para facilitar a visualização, o ponto de referência do sistema é movido para o centro da tela, o primeiro objeto é escalado por um fator de 200.0 e o segundo é escalado em 100.0 e rotacionado em 45 graus no sentido anti-horário.

B.5 Visualizando a cena de colisão

Outra *feature* da linguagem Colli é possibilidade visualizar a disposição espacial dos objetos descritos, sendo possível assim verificar por inspeção se o resultado dos testes de colisão está correto. Para isso, basta utilizar o comando `show scene` em qualquer

[2] A atual versão do interpretador não possui suporte a objetos côncavos, e testes com estes tipos de objeto tem resultado indefinido.

momento de um *script* Colli. Este comando irá gerar uma representação visual do código até aquele momento, travando a execução do interpretador.

O visualizador gera uma tela de tamanho 800x600 *pixels*, e considera que a dimensão de cada objeto também é dada em *pixels*. Dessa forma, *bounding boxes* de dimensões pequenas não são mostrados adequadamente. Entretanto, é importante notar que os testes de colisão ainda são válidos para estes *bounding boxes*, tendo apenas a visualização prejudicada. Os objetos são desenhados com base em coordenadas absolutas.

O Código B.5 mostra o uso do visualizador em dois momentos distintos, e os resultados de tais chamadas são mostrados nas Figuras 15 e 16.

Código B.5 – Invocando o visualizador.

```
1 reference is 400.0, 300.0
2
3 first is bound by circle scaled 100.0
4 second is bound by square scaled 100.0 translated -100.0, 0.0 rotated
      45.0
5
6 show scene
7
8 second is translated 200.0, 0.0
9
10 show scene
```

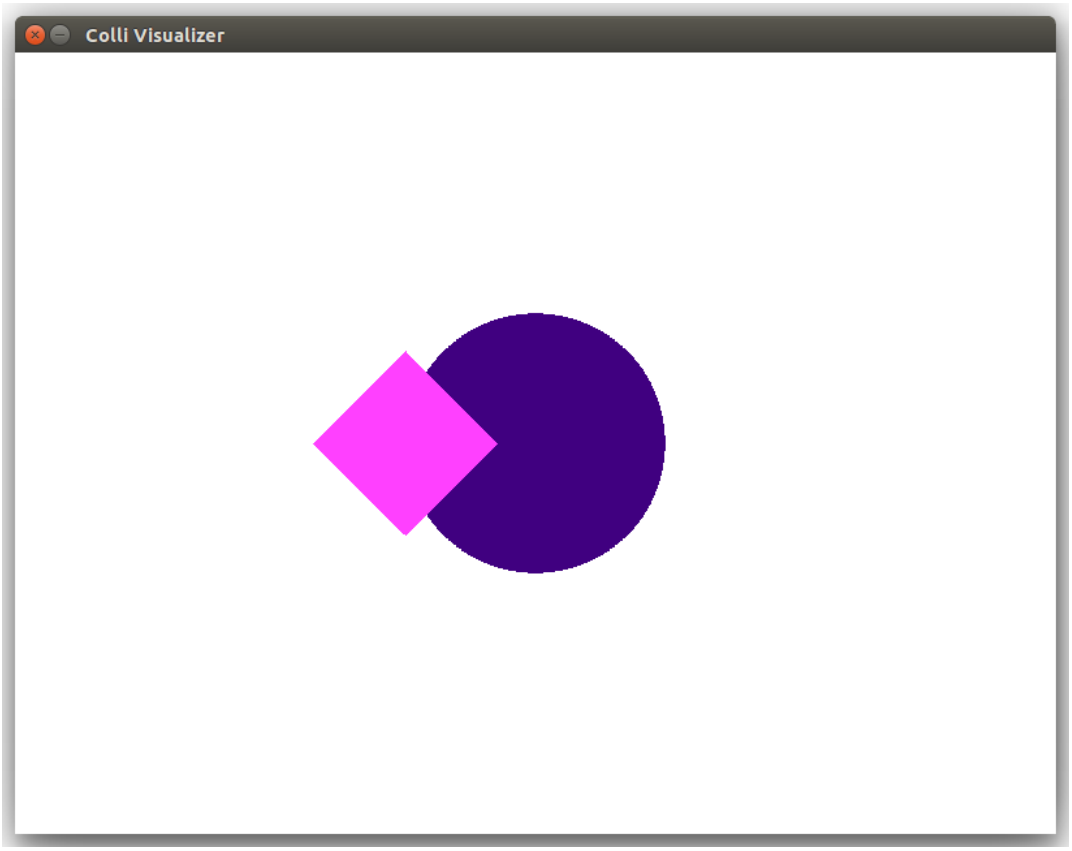

Figura 15 – Cena de colisão antes da translação.

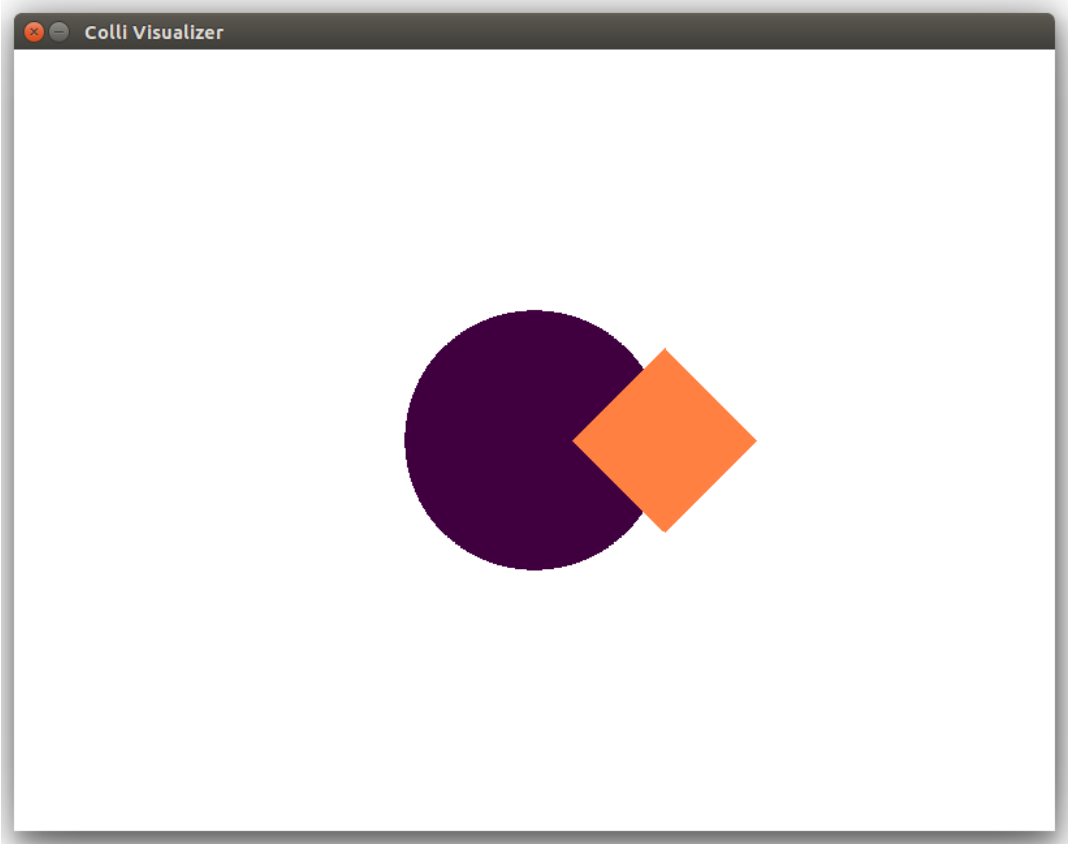

Figura 16 – Cena de colisão após a translação.

B.6 Integração com a Linguagem C

Colli ainda dispõe de uma API de integração com a linguagem C. Esta API utiliza uma pilha para comunicar-se com o interpretador, sendo que cabe ao usuário empilhar os valores apropriados para realizar a operação que deseja. As funções de controle estão disponíveis no arquivo *colli.h*. Para ser capaz de utilizar a API em seu código C, o usuário deve inicializar a pilha através da função `colliInit()`. Esta função retorna um ponteiro do tipo *ColliStack* que servirá como parâmetro de controle para as demais funções.

A pilha suporta quatro tipos de valores:

- **Comando**: Na pilha, este valores representam os comandos da linguagem, sendo colocados na mesma através da função `colliPushCommand`, que recebe como parâmetro um inteiro. Para aumentar a clareza do código, os valores possíveis foram declarados como parte da enumeração `StackDefs`.
- **Identificador**: Representa o nome dado a um dos objetos. Estes valores são colocados na pilha diretamente na forma de `string`.
- **Valor**: Representa os valores para rotação, contração/expansão linear e translação. Estes valores devem ser enviados como `double`, para serem colocados na pilha.
- **Forma**: Estes valores representam os possíveis formatos dos *bounding boxes*. Os valores passados devem ser `strings` como `"circle"`, `"square"` ou uma lista no formato esperado pelo interpretador Colli (isto é, `"{0.0, 0.0; 1.0, 1.0; 1.0, 0.0}"`).

Este valores são colocados na pilha utilizando as funções `colliPushCommand()`, `colliPushId()`, `colliPushValue()` e `colliPushShape()`, respectivamente. Uma vez empilhados os valores, o usuário deve chamar a função `colliRun()`. Esta função verifica se o parâmetro no topo da pilha é um comando, buscando os parâmetros apropriados na pilha para executar automaticamente. Entre os comandos Colli, apenas as requisições de teste de colisão fogem deste padrão, sendo realizadas utilizando a função `colliTest()`, que recebe como parâmetro o identificador sob o qual deseja-se realizar o teste de colisão, e retorna um vetor de `strings` que contém os identificadores com os quais houve colisão.

De forma geral, os valores não necessitam ser empilhados em uma ordem específica, exceto quando deseja-se executar uma operação de translação, na qual deve-se colocar na pilha primeiro o valor do *offset* em x, e em seguida o valor do *offset* em y. O Código B.6 mostra como proceder para criar dois objetos e realizar um teste de colisão entre os mesmos.

Código B.6 – Utilização de Colli na linguagem C.

```
#include <stdio.h>
#include <stdlib.h>

#include "colli.h"

int main(){

        int i = 0;
        char **results = NULL;
        int totalCollisions = 0;

        ColliStack *s = colliInit();

        colliPushId(s, "first");
        colliPushShape(s, "circle");
        colliPushCommand(s, BIND);
        colliRun(s);

        colliPushId(s, "first");
        colliPushCommand(s, SHOW);
        colliRun(s);

        colliPushId(s, "second");
        colliPushShape(s, "circle");
        colliPushCommand(s, BIND);
        colliRun(s);

        colliPushId(s, "second");
        colliPushValue(s, -0.5);
        colliPushValue(s, 0.5);
        colliPushCommand(s, TRANSLATE);
        colliRun(s);

        colliPushId(s, "second");
        colliPushCommand(s, SHOW);
        colliRun(s);

        results = colliTest("first", &totalCollisions);
```

```
        if(results){
                for(i = 0; i < totalCollisions; i++)
                        printf("first␣collided␣with␣%s\n", results[i]);
        }

        return 0;
}
```

A função `colliTest()` irá retornar um *array* de `strings` contendo os identificadores dos objetos que estão colidindo com o objeto passado como parâmetro. No Código B.7 são mostrados os comandos equivalentes em sintaxe Colli para executar o teste do Código B.6.

Código B.7 – Código Colli equivalente.

```
first is bound by circle

second is bound by circle
second is translated -0.5, 0.5

collisions with first
```

Em ambos os casos, o resultado mostrado na tela deve ser o texto `first collided with second`

B.7 Conclusão

Colli é uma linguagem com um cojunto sintático reduzido, buscando facilitar a compreensão por parte do usuário. Utilizando os comandos para escalar, rotacionar e transladar os objetos criados permite versatilidade na descrição de *bounding boxes*, e, por se tratar de uma linguagem de *script*, mudanças em sua estrutura não requerem recompilação do código fonte, auxiliando no processo de desenvolvimento.