# Detectarea de coliziuni între cercuri

Universitatea Transilvania din Brașov

June 24, 2022

## 1 Coliziune

### 1.1 Definiție și paradigmă

Când aducem în discuție termenul de coliziune, ne referim la ciocnirea de forțe sau la tangența dintre obiecte care se mișcă unul spre altul.

### 1.2 Context informatic

În ceea ce privește domeniul informatic, importanța deosebită pe care o are un sistem de detectare a coliziunilor poate fi pusă în evidență în simulări pentru sisteme fizice, de exemplu, sau în cadrul jocurilor video, unde detectarea de coliziuni și rezolvarea lor implică o cărămidă solidă în obținerea unui produs ce poate fi, în primul rând, jucat.

## 2 Eficiență

### 2.1 Probleme

Alegând arbitrar un spațiu dreptunghiular și 100 de obiecte care îl populează, putem spune că, pentru verificarea tuturor coliziunilor dintre obiectele , este necesar să realizăm aproximativ 10000 de operații, fiecare obiect cu fiecare dintre celelalte din jur. Acest fapt aduce cu sine problema de eficiență a verificării coliziunilor în timp pătratic. Imaginându-ne că într-un joc video, luând exemplul anterior, numărul de obiecte cu care jucătorul interacționează, dar și numărul altor entități care se deplasează și necesită operația de verificare a coliziunilor, depășește rapid ordinul miilor, astfel că o soluție pătratică nu poate fi pusă în practică fără pierderi masive în performanță.

### 2.2 Soluții

O soluție comună în rândul programatorilor o constituie partiționarea spațiului inițial. Motivul din spatele acestei metode de diminuare a numărului de operații este intuitiv: două obiecte aflate la capete opuse ale unui spațiu sau sub-spațiu nu se ating, drept urmare, nu este necesară operația de verificare între obiecte aflate la distanțe mari unul de celălalt. Această separare a spațiului se poate realiza folosind o structură de de tip *quadtree*.

## 3 Quadtree

### 3.1 Definiție și paradigmă

Un *quadtree* este o structură de care codifică un spațiu bidimensional în celule adaptabile sau ajustabile din punct de vedere al mărimii. Asemănător cu arborii binari, *quadtrees* sunt arbori în care fiecare nod care nu reprezintă o frunză va avea patru copii.

### 3.2 Partiționare prin cadrane

În general, un *quadtree* este reprezentat grafic asemenea unei tabele, fiecare celulă a acesteia reprezentând unul dintre noduri. Orice *quadtree* începe cu un singur nod, urmând ca mai apoi să se adauge mai multe pe măsură ce crește numărul de obiecte. La partiționare, fiecare obiect va fi încadrat în unul dintre cei 4 copii ai nodului curent. Orice obiect care nu se încadrează complet în interiorul unui nod, sau careu, va fi plasat în nodul părinte zonei nou partiționate. Fiecare nod, sau careu, se va subdiviza la rândul său, pe măsură ce sunt adăugate obiecte.

## 4 Implementarea arborelui

### 4.1 Câmpuri și metode folosite

În ceea ce privește detaliile de implementare, arborele cuprinde drept câmpuri private un număr maxim de obiecte, o variabilă booleană care reține dacă un nod se împarte la rândul său, un vector de noduri, un vector de cercuri și un dreptunghi, definit de coordonatele unui colț și lungimile laturilor. Metodele folosite sunt cele de inserare, ștergere, împărțire, ștergere completă și de afișare grafică. De menționat este faptul că dreptunghiul cuprinde și o metodă care verifică dacă un cerc arbitrar ales este sau nu aflat în interiorul lui.

### 4.2 Constructori

Constructorul *quadtree*-ului setează un număr maxim de obiecte egal cu 10, inițializează cu *false* valoarea variabilei care ține cont dacă un nod este sau nu împărțit în sub-noduri și desemnează coordonatele primul dreptunghi.

```
quadtree::quadtree(Rectangle rectangle)
{
        this->isSplit = false;
        this->boundaries = rectangle;
        this->MAX_OBJECTS = 10;
}
```

Constructorul dreptunghiului pune implicit coordonatele colțului pe poziția (0; 0) și consideră lungimea ambelor laturi egală cu 1.

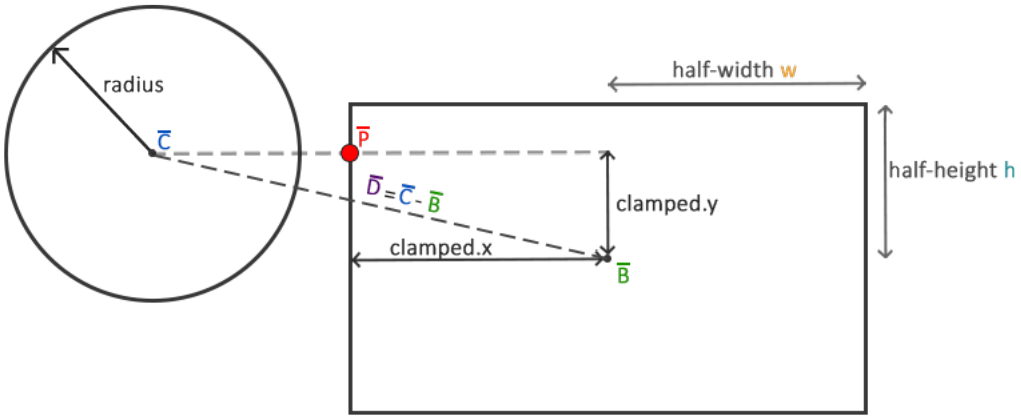
```
Rectangle(double coordX = 0, double coordY = 0, double width = 1, double height = 1)
{
        this->coordX = coordX;
        this->coordY = coordY;
        this->width = width;
        this->height = height;
}
```

### 4.3 Metode

#### 4.3.1 Verificarea dacă un cerc este într-un dreptunghi

Pentru realizarea acestei funcții, am folosit drept fundament matematic $AABB$-ul, abreviere pentru *axis aligned bounding box*. Pe scurt, am calculat coordonatele punctelor care împart laturile dreptunghiului curent în jumătate, iar apoi coordonatele centrului acestuia. Am salvat, ulterior, diferența dintre coordonatele centrului dreptunghiului și cele ale cercului dat ca parametru, definit prin coordonatele centrului și lungimea razei. Urmează să salvăm și diferența, pe coordonate, dintre centrul cercului dat și centrul dreptunghiului curent. Mai departe, am folosit ceea ce se numește *clamping*, metodă ce restricționează o valoare numerică într-un interval dat de alte două. În cazul de față, folosim această metoda a afla cel mai apropiat punct al dreptunghiului de centrul cercului. Funcția va returna dacă distanța euclidiană de la cel mai apropiat punct al dreptunghiului de cerc este mai mica sau egala cu raza.

```
#define euclidianDistance(x, y) (((x)*(x)) + ((y)*(y)))
#define clamp(x, y, z) std::max((y), std::min((z), (x)))
```




```
bool checkCircleBounds(double cx, double cy, double radius)
{
    double halfXAxis = this->width / 2;
        double halfYAxis = this->height / 2;

        double centerX = this->coordX + halfXAxis;
        double centerY = this->coordY + halfYAxis;

        double differenceX = cx - centerX;
        double differenceY = cy - centerY;

        double clampedX = clamp(differenceX, -halfXAxis, halfXAxis);
        double clampedY = clamp(differenceY, -halfYAxis, halfYAxis);

    double closestX = centerX + clampedX;
        double closestY = centerY + clampedY;

        differenceX = closestX - cx;
        differenceY = closestY - cy;

        return euclidianDistance(differenceX, differenceY) <= radius * radius;
}
```

### 4.3.2 Inserare

Tocmai pentru eficientizarea prin partiționarea spațiului, prima verificare, înainte de a realiza orice inserare, se rezumă la a vedea dacă cercul de inserat intră sau nu în dreptunghiul curent.

În caz afirmativ, putem verifica dacă nodul curent nu a fost împărțit până la acest moment, iar sub-arborele admite încă inserări, nedepășind limita superioară menționată drept câmp privat, mai sus. Dacă aceste condiții sunt respectate, se poate pune cercul curent în vectorul de cercuri și se poate realiza verificarea dacă cercul curent se intersectează cu oricare altul din sub-arborele în care se află. Această verificare se folosește de distanța euclidiana dintre centrele cercurilor verificate. Dacă aceasta este mai mică sau egala decât suma razelor cercurilor analizate, înseamnă că cercurile pot fi adăugate într-un vector de perechi de cercuri care realizează coliziune.

În cazul în care condițiile enumerate în paragraful anterior nu sunt respectate, se realizează o împărțire a sub-spațiului, iar fiecare nod va apela recursiv funcția de inserare pentru a se reîncerca distribuirea sub-spațiilor.

```
void quadtree::insertQ(circle* currentCircle,
                       std::vector<std::pair<circle*, circle*>>& collidingCircles)
{
        if (boundaries.checkCircleBounds(currentCircle->coordx,
                                         currentCircle->coordy,
                                        currentCircle->radius) == false)
                return;
        if (isSplit == false && circles.size() < MAX_OBJECTS)
        {
                circles.push_back(currentCircle);
```

```
                for (auto anyCircle : circles)
                {
                        if (currentCircle != anyCircle &&
                                ((currentCircle->coordx - anyCircle->coordx) *
                                (currentCircle->coordx - anyCircle->coordx)) +
                                ((currentCircle->coordy - anyCircle->coordy) *
                                (currentCircle->coordy - anyCircle->coordy)) <=
                                (currentCircle->radius + anyCircle->radius) *
                                (currentCircle->radius + anyCircle->radius))
                                collidingCircles.push_back({currentCircle, anyCircle});
                }
        }
        else
        {
                if (isSplit == false)
                        split(collidingCircles);
                for (auto node : nodes)
                        node->insertQ(currentCircle, collidingCircles);
        }
}
```

### 4.3.3 Împărțire

Împărțirea se realizează cunoscând coordonatele sub-spațiului curent. Cu ajutorul lor, putem obține lungimea și lățimea noului sub-spațiu, egale cu jumătate din cele ale sub-spațiului acum partiționat. Urmează a se pune în vectorul de noduri patru noi noduri, care reprezintă la rândul lor dreptunghiuri.

În final, se inserează din nou cercurile, pentru fiecare nod, se șterg elementele din vectorul de cercuri, iar spațiul curent este considerat acum partiționat.

```
void quadtree::split(std::vector<std::pair<circle*, circle*>>& collidingCircles)
{
        double x = boundaries.coordX;
        double y = boundaries.coordY;

        double subWidth = boundaries.width / 2;
        double subHeight = boundaries.height / 2;
                                                                // Bottom Left
        nodes.push_back(std::shared_ptr<quadtree>(new quadtree(Rectangle(x,
                                                                        y + subHeight,
                                                                        subWidth,
                                                                        subHeight))));
                                                                // Bottom Right
        nodes.push_back(std::shared_ptr<quadtree>(new quadtree(Rectangle(x + subWidth,
                                                                        y + subHeight,
                                                                        subWidth,
                                                                        subHeight))));
                                                                // Top Right
        nodes.push_back(std::shared_ptr<quadtree>(new quadtree(Rectangle(x + subWidth,
                                                                        y,
                                                                        subWidth,
                                                                        subHeight))));
                                                                // Top Left
        nodes.push_back(std::shared_ptr<quadtree>(new quadtree(Rectangle(x,
                                                                        y,
                                                                        subWidth,
                                                                        subHeight))));

        for (auto currentObject : this->circles)
                for (int index = 0; index < nodes.size(); index++)
                        nodes[index]->insertQ(currentObject, collidingCircles);
        this->circles.clear();
        this->isSplit = true;
}
```

### 4.3.4 Ștergere

Ștergerea începe identic cu inserarea, întrucât este necesar sa cunoaștem dacă cercul este sau nu încadrat în nodul, sau sub-spațiul, curent. Dacă sub-spațiul curent nu mai este partiționat, sau în alți

termeni, dacă nodul curent este frunză, atunci putem parcurge cercurile prezente în acel sub-spațiu și îl putem șterge pe cel dorit. Altfel, dacă spațiul curent este partiționat, continuăm căutarea pe alt sub-nod.

```
void quadtree::deleteQ(circle* currentCircle)
{

        if (boundaries.checkCircleBounds(currentCircle->coordx,
                                         currentCircle->coordy,
                                         currentCircle->radius) == false)
                return;

        if (isSplit == false)
        {
                for (int index = 0; index < circles.size(); index++)
                {
                        if (circles[index] == currentCircle)
                        {
                                this->circles.erase(circles.begin() + index);
                                return;
                        }
                }
        }
        else
        {
                for (auto node : nodes)
                        node->deleteQ(currentCircle);
        }
}
```

### 4.3.5 Ștergere completă

Dacă avem un sub-spațiu partiționat, apelăm funcția de ștergere completă recursiv, până la întâlnirea primei frunze, sau a primului sub-spațiu copil al spațiului de la care am pornit. Pentru acesta, ștergem toate nodurile și toate cercurile din interior, după care, la reapel, după ștergerea tuturor celor patru copii, nodul curent devine nepartiționat.

```
void quadtree::clear()
{
        if (isSplit == true)
        {
                for (auto node : nodes)
                        node->clear();
                isSplit = false;
        }
        nodes.clear();
        circles.clear();
}
```

### 4.3.6 Afișare grafică

Pentru *output*-ul grafic am folosit biblioteca : olcPixelGameEngine.h, ce permite cu ușurință să desenăm figuri geometrice, linii și segmente sau să scriem text. În contextul acestei funcții, am afișat pentru fiecare nod spațiul pe care îl acoperă, precum și numărul de cercuri din fiecare careu.

```
void quadtree::graphical_output(olc::PixelGameEngine* instance)
{
        if (isSplit == false)
                instance->DrawString(boundaries.coordX + 2, boundaries.coordY + 2,
                                     std::to_string(this->circles.size()), olc::MAGENTA);
        else
        {
                for (auto node : nodes)
                        node->graphical_output(instance);
        }
        instance->DrawRect(boundaries.coordX, boundaries.coordY,
                           boundaries.width, boundaries.height);
}
```

# 5 Implementarea obiectelor

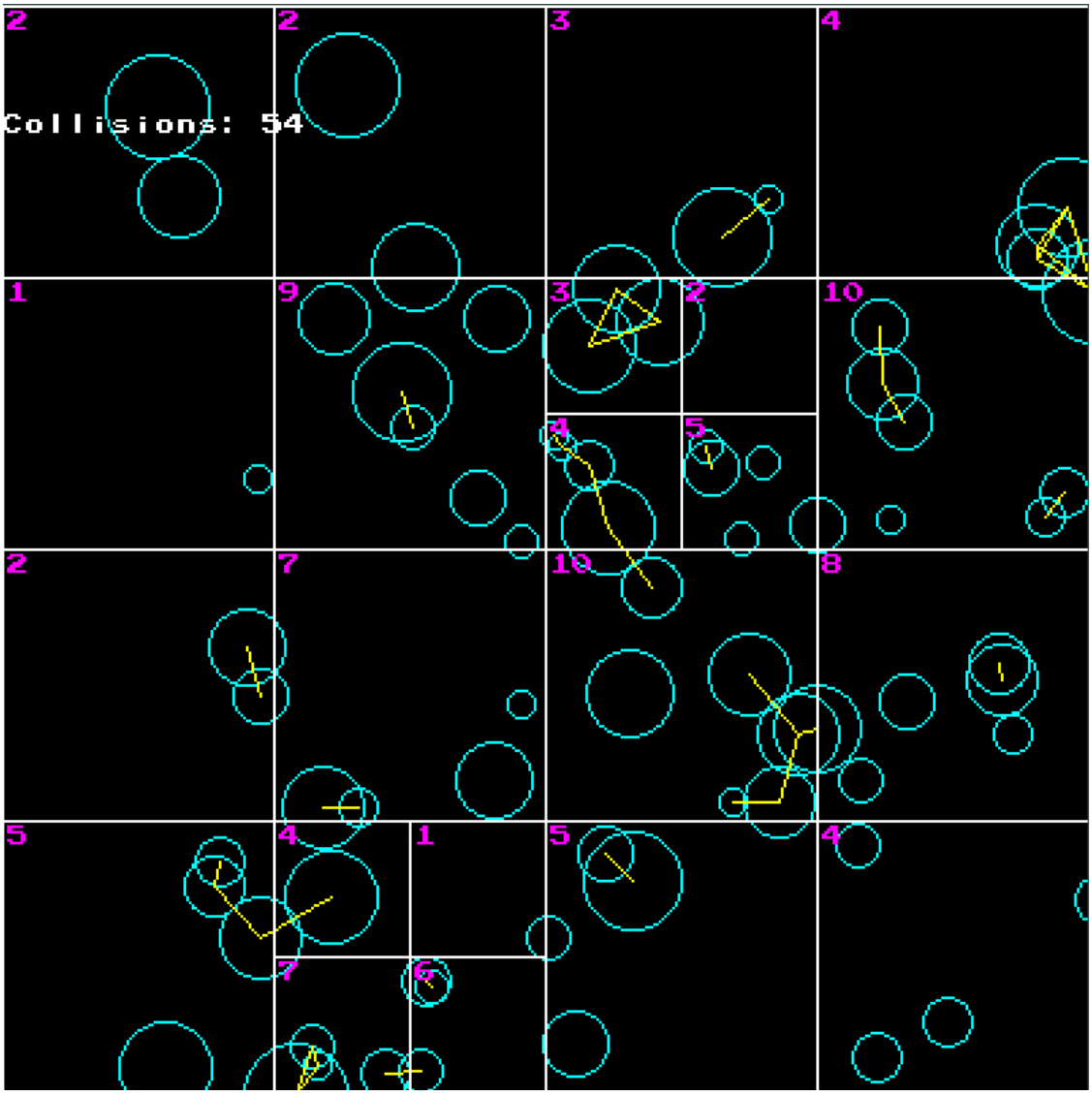
Orice obiect deriva din clasa *circle.h* care are ca și câmpuri private coordonatele centrului, viteză, accelerație, unghi și rază. Obiectele traversează ecranul de la dreapta la stânga, pornind de la o coordonata aleatorie de pe ecran, cu o viteză intre 1 și 15 și unghi inițial 0. O variabilă ce va ține minte rotația va defini unghiul în funcția de *update*. La viteza inițială, la fiecare apel al funcției de *update*, se adună timpul înmulțit cu accelerația. Coordonatele centrului se actualizează și ele, împreună cu unghiul, iar accelerația devine 0.

```
void object::createObject(int noOfVertices)
{
        for (int index = 0; index < noOfVertices; index++)
        {
                double a = ((double)index / (double)noOfVertices) * 6.28318;
                vectorOfVertices.push_back({radius * sin(a), radius * cos(a)});
        }
}

void object::updateObject(float time)
{
        this->velocityX += this->accelerationX * time;
        this->velocityY += this->accelerationY * time;

        this->coordx += this->velocityX * time;
        this->coordy += this->velocityY * time;

        this->angle += this->spin * time;

        this->accelerationX = 0;
        this->accelerationY = 0;
}
```

# 6 Aplicație

## 6.1 Prezentare

Aplicația urmărește punerea în evidență a coliziunilor dintre cercuri care traversează ecranul de la stânga la dreapta, fiind vizibile totodată spațiile partiționate și numărul de cercuri care le populează. Între oricare două cercuri tangente se trasează o linie, pentru ilustrarea coliziunii.




## 6.2 Implementare

Funcțiile de *OnUserCreate()* și *OnUserUpdate()* fac parte din biblioteca olcPixelGameEngine.h și se referă la generarea condițiilor de start ale aplicației și, respectiv, continuarea și alterarea condițiilor în ciclu infinit. Pentru a modifica în clasa aplicației aceste metode, am utilizat funcționalitatea de *override* din C++.

Funcția de *OnUserCreate()* creează primul arborele cu nodul inițial și resetează datele: șterge toate obiectele, distruge toate nodurile, realizează un *random seed* și generează 100 de obiecte noi, pe care le adaugă în vectorul de obiecte și le inserează în arbore.

```
bool application::OnUserCreate()
{
        QTree = new quadtree(quadtree::Rectangle(0, 0, ScreenWidth(), ScreenHeight()));
        resetApp();
        return true;
}
```

Funcția de *OnUserUpdate()* setează fundalul negru, șterge elementele din vectorul care reține elementele aflate la un moment dat în coliziune, șterge arborele și, pentru fiecare obiect din șirul de obiecte, se realizează o inserare în arbore, o deplasare a coordonatelor și un apel al funcției DrawCircle() pentru afișarea grafică a cercului. Se scrie, totodată, pe ecran, numărul de elemente aflate în coliziune, iar pentru fiecare pereche de obiecte din șirul de obiecte tangente, se desenează o linie galbenă între centrul primului cerc și centrul cercului de-al doilea.

```
bool application::OnUserUpdate(float time)
{
        Clear(olc::BLACK);
        collidingCircles.clear();
        QTree->clear();
        for (auto currentObject : objects)
        {
                QTree->insertQ(currentObject, collidingCircles);
                currentObject->updateObject(time);
                WrapCoordinates(currentObject->coordx,
                                currentObject->coordy,
                                currentObject->coordx,
                                currentObject->coordy);
                DrawCircle(currentObject->coordx,
                           currentObject->coordy,
                           currentObject->radius,
                           olc::CYAN);
        }
        DrawString(0, 40,
                   "Objects colliding: " + std::to_string(collidingCircles.size()));
        for (auto currentObject : collidingCircles)
                DrawLine(currentObject.first->coordx,
                         currentObject.first->coordy,
                         currentObject.second->coordx,
                         currentObject.second->coordy,
                         olc::YELLOW);
        QTree->graphical_output(this);

        return true;
}
void application::WrapCoordinates(double inputX, double inputY,
                                  double outputX, double outputY)
{
        outputX = inputX;
        outputY = inputY;

        if (inputX < 0)
                outputX = inputX + (double)ScreenWidth();
        if (inputX >= (double)ScreenWidth())
                outputX = inputX - (double)ScreenWidth();
        if (inputY < 0)
                outputY = inputY + (double)ScreenHeight();
        if (inputY >= (double)ScreenHeight())
                outputY = inputY - (float)ScreenHeight();
}
```

# 7 Posibilități de expansiune

Orice proiect are o posibilitate de a fi continuat spre a atinge noi limite de performanță sau noi funcționalități. Proiectul discutat în această lucrare poate fi extins pe mai să rezolve problemele ce apar la coliziune, prin introducerea unui sistem de forțe fizice. Totodată, se pot realiza operații de verificare a coliziunii pentru orice forma geometrică. Sistemul creat poate fi folosit cu ușurință și în realizarea unui joc video de tip arcade, precum un *space shooter* sau *PAC-MAN*.

# 8 Concluzii

Câteva dintre concluziile pe care le putem extrage se referă la eficiența acestei metode de analizare a coliziunilor. Putem realiza rapid că numărul de careuri influențează numărul de operații de verificare. Dacă numărul de careuri este mai mic, atunci este folosită mai puțină memorie. Implicit, în această situație, numărul de verificări este mai mare. Invers, dacă numărul de careuri este considerabil mai mare, numărul de operații de verificare scade, dar memoria consumată este mai mare. Un dezavantaj al împărțirii pe foarte multe careuri îl reprezintă numărul mare de noduri care sunt goale, adică nu cuprind niciun cerc, întrucât acestea reprezintă o risipă de memorie.

O altă concluzie extrasă se referă la modalitățile prin care se pot realiza aceste probleme de coliziuni. *Quadtree*-ul nu reprezintă singura structură de date eficientă pentru gestionarea partiționării spațiului bidimensional. O variantă alternativă o reprezintă un *K-D tree* care partiționează neuniform, sortând elementele după una dintre axe și trasând o axă mediană intre cele două obiecte aflate la mijlocul șirului sortat. Procedeul se repetă și pentru axa opusă și se va termina atunci când este atins numărul maxim de obiecte dintr-un careu sau, mai eficient, când cat mai multe careuri cuprind un singur obiect, astfel încât pentru multe dintre acestea să nu se mai realizeze operația de verificare.

Se poate aminti, totodată, faptul că problema coliziunilor rămâne una de importanță deosebită cu cât numărul de cadre pe secunda, sau *FPS*, este mai mic. În detectarea coliziunilor, cât și în rezolvarea lor, dacă viteza unui obiect este foarte mare și *FPS*-ul mic, se poate ca obiectul în mișcare, la următoarea etapă a procesului de *rendering*, să iasă din limitele prestabilite și să intre într-un alt obiect sau să treacă prin el fără detectarea coliziunii. Astfel de probleme țin deja de etapa propriu-zisă de *rendering*, dar eficiența sistemului de *rendering* influențează direct calitatea calculului de coliziune.

Un alt aspect extras din procesul de documentare, împreună cu lucrul propriu-zis la acest proiect, îl reprezintă fundamentele matematice pe care se bazează detectarea de coliziuni. Dacă între cercuri sau poligoane convexe funcționează cu succes calcularea proiecțiilor punctelor pe axe și verificarea dacă acestea se încadrează în intervalul punctelor definit de unul dintre ele sau calculul intersecției diagonalelor, la poligoane concave apar numeroase probleme mai greu de rezolvat.

# 9 Referințe

Pentru implementarea structurii de date, am folosit informațiile prezentate în următorul articol: Quick Tip: Use Quadtrees to Detect Likely Collisions in 2D Space

Pentru informațiile despre *K-D tree*, am folosit drept suport următorul videoclip: Building Collision Simulations: An Introduction to Computer Graphics

Suportul matematic discutat provine din videoclipul următor: Convex Polygon Collisions #1 și din articolul: Collision detection