# GameEngine - ReadMe

Programmering i C och C++
Institutionen för Data- och Systemvetenskap
Stockholms Universitet

Elias Bennaceur
Alexander Jaxgård

# Bakgrund

Detta projekt strävar efter att erbjuda följande funktioner och uppfylla specifika mål:

- Effektiv och resursvänlig 2D-rendering.
- Enkel integration och användning.
- Stöd för olika plattformar.
- Anpassningsbarhet och utbyggbarhet.

## Requirements

För att skapa ett spel med denna spelmotor krävs följande:

- En integrerad utvecklingsmiljö
    - Exempel på detta är Microsoft Visual Studio Code, Xcode, Visual Studio
- Grundkunskap inom C++ och C
    - Eftersom att spelmotorn är skriven i C++ kan det vara hjälpsamt att ha erfarenhet och grundkunskap inom språket.

# Att börja skapa ett spel

## Filsystemet

Detta projekt har följande filsystem:

- .vscode/
  - c_cpp_properties.json
  - launch.json
  - settings.json
  - tasks.json
- build/
  - /debug/
    - /play.dSYM/
      - /resources/
        - /DWARF/
          - play
        - /relocations/
          - /x86_64/
            - play.yml
    - play
    - play.exe
- include/
  - Constants.h
- resources/
  - fonts/
  - images/
  - sounds/
  - spritesheet/
- src/
  - main_game.cpp
  - Component.cpp
  - Component.h
  - Session.cpp
  - Session.h
  - System.cpp
  - System.h

Gör dig bekväm med detta filsystem och kolla grundligt igenom vad de olika filerna har för uppgift i spelmotorn. Var beredd på att spelet använder biblioteket

SDL (Simple DirectMedia Layer). Du kan läsa mer om SDL och vad det innebär att använda sig av SDL på deras webbplats: www.libsdl.org.

## Session.cpp

Session.cpp är den fil som håller koll på och uppdaterar dina sprites och objekt genom en konstant funktion draw() i Component.h som inte kan ändras, men även genom tick() som kan ändras.

Session.cpp använder sig av en add()-funktion för att lägga till objekt med typen Component, samt en remove()-funktion som tar bort ett objekt.

Genom main_game.cpp kan du nå både add() och remove() efter skapande av ett Session-objekt. Glöm inte att #include <Session.h>.

## System.cpp

System.cpp är den mest väsentliga filen, däremot är det den fil som du inte kommer att jobba med lika mycket. Den startar SDL och kör händelseloop för att spelet ska kunna starta.

Du kan kalla på olika funktioner för att t.ex få din renderare och spela ljud. Eftersom att System är deklarerad globalt som en variabel sys, kan du anropa dess funktioner direkt i main_game eller dina egna spelklasser.

För att spela ett ljud genom System anropar du funktionen på följande sätt:
cwing::System::play_sound(“ditt ljud.filtyp”);

För att få din renderare anropar du
cwing::System::get_ren();

## Dina egna spelklasser/Component.cpp

Som tidigare nämnt används en draw()-funktion och en tick()-funktion för att hålla koll på och uppdatera dina objekt. Component.h definerar dock ett antal andra virtuella funktioner, som du måste inkludera i dina egna spelfiler som är direkta subklasser till Component.

Dessa virtuella funktioner är:

- virtual void mouseDown(const SDL_Event &eve) {}
  - Denna funktion anropas av Session när musknappen trycks ner av användaren.
- virtual void mouseUp(const SDL_Event &) {}
  - Denna funktion anropas av Session när användaren släpper musknappen.
- virtual void keyDown(const SDL_Event &) {}
  - Denna funktion anropas av Session när ett knapptryck registreras.
- virtual void keyUp(const SDL_Event &) {}
  - Denna funktion anropas av Session när knappen släpps.
- virtual void mouseMotion(const SDL_Event &) {}
  - Denna funktion anropas av Session när användaren rör på musen.
- virtual void draw() const = 0;
  - Denna funktion anropas av Session när objektet ska ritas ut. Detta sker oftast för varje frame. Oftast görs inte draw() direkt i objektet, utan istället anropas Components draw, genom cwing::Component::draw().
- virtual void tick() = 0;
  - Denna funktion anropas av Session varje frame. Denna funktion kan anropas för att röra på objekt eller annat.
- virtual void collision(Component *comp) = 0;
  - Denna funktion anropas av Session när ett objekt kolliderar med ett annat.
- virtual std::string getLabel() { return label; }
  - Denna funktion anropas av olika objekt som har implementerat collision(Component *comp), för att anpassa händelsen till objektet den kolliderar med. Denna funktion kan dock användas till annat.

Genom tick() i din klass kan du skapa anpassade rörelser och händelser för ditt objekt.

```
void Enemy::tick()
{
    // moveForward();

    if (getRect().x + getRect().w < 0 || getRect().x > ses.getScreenWidth() ||
        getRect().y + getRect().h < 0 || getRect().y > ses.getScreenHeight())
    {
        ses.remove(this);
    }
}
```

```
void Enemy::moveForward()
{
    rectangle.y += velocity;
    Component::setRect(rectangle.x, rectangle.y, rectangle.w, rectangle.h);
}
```

Ovan visar hur tick() är implementerad i en potentiell Enemy-klass, där objektet tas bort genom sessions egna remove-funktion.

## Main_game

I main_game skapar du ditt spel. Här skapar du egna objekt av klasser som du skapar i src/.

```cpp
int main(int argc, char **argv)
{

    std::cout << "*** main()\n";

    Session ses;
    std::clog << "Session Started" << std::endl;
    ImageComponent *backgroundImage = new ImageComponent(0, 0, ses.getScreenWidth(), ses.getScreenHeight(), "bg.bmp");


    Player *player = new Player(ses.getScreenWidth()/2, ses.getScreenHeight()-(ses.getScreenHeight()/4), 32, 32, "images/rymdskepp.bmp", ses);
    ses.add(backgroundImage);
    ses.add(player);
    Enemy *enemy = new Enemy(ses.getScreenWidth()/2, 0, 25, 25, 1, ses);



    ses.add(enemy);




    ses.run();

    std::clog << "session running" << std::endl;

    return 0;
}
```

Ett tips är att inte skriva klasser i main_game, utan att istället dela upp spel- och andra klasser i separata cpp- och headerfiler. Detta kommer att skapa struktur i ert spel och göra det lättare att hålla koll på vad som gör vad.

# Att bygga och köra spelet

1. Skapa klasser i olika filer såsom t.ex Player.cpp och Player.h, Enemy.cpp och Enemy.h
    a. Dessa klasser ska vara subklasser till Component om det är klasser som ska inkluderas på skärmen.
        i. När du skapar subklasser till Component kan du inkludera “Constants.h” och använda Constants::gResPath + “/images/dinbild.bmp” för att lokalisera din sprite. Detta gör att du slipper skriva in den fullständiga adressen när du letar. Du kan på samma sätt lokalisera annat som behövs till ditt spel. Du byter då ut images till antingen fonts, sounds eller spritesheets.
2. För att få en bakgrundsbild behövs en bildkomponent som inte ska gå att styras på. Du kan göra detta genom att skapa en ny klass.
3. För att ändra bakgrundsljud i form av musik ändrar du rad 24 i System.cpp.
4. I din klass behöver du definiera ett antal funktioner som är virtuella hos Component. Detta innebär att du måste definiera och implementera dessa på dina subklasser. Notera att dessa implementationer kan vara tomma, ifall du inte använder dem.
    a. Se Dina egna spelklasser/Component.cpp för att veta mer om vilka dessa virtuella funktioner är och vad de gör i förhållande till spelet.
5. När du väl har skapat dina klasser kan du skapa instanser av dessa och lägga till dem till din Session.
    a. Du instansierar en Session genom att inkludera “Session.h” i _main_game och gör en ny Session genom att skriva “Session *ses = new Session(). Denna kan du sedan använda i ditt spel.
    b. Du lägger till dina instanser genom att skriva ses.add(dittObjekt)
        i. Se till att byta ut “dittObjekt” med namnet på ditt faktiska objekt.
6. Till slut skriver du ses.run() i _main_game för att se till att din session körs.