START A DOT

游戏设计报告

目录

一、初始灵感.....3
二、设计.....4
游戏逻辑设计与游戏完整流程.....4
视觉设计.....4
UI 设计.....4
字体设计与选用.....4
音乐设计.....6
交互设计.....6
三、软件工程.....7
编程 API 选择.....7
在软件工程方面考虑.....7
游戏引擎架构.....7
游戏引擎整体架构.....7
ROC Controller.....8
ROC Studio.....9
ROC Level.....10
ROC Object.....10
ROC Sprite.....11
无限生成的数据结构.....11
一些具体算法.....12
球与直线的碰撞反弹.....12
球与球的反弹.....12
难度递增算法.....12

一、初始灵感

对于一个非常小型的游戏，比较重要的一点是让玩家非常快速的上手这一款游戏，在适应了基础的游戏规则后，游戏也可以提供一些道具和技能之类的附加玩法。这种模式同样对游戏开发非常重要，首先建立起基础的游戏规则和玩法，然后再去添加附加的东西，使得最初的游戏开发专注于核心玩法的构建，保证核心玩法的游戏性。

基于第一节课弹球的灵感，选择使用一个小球为主角，开发了两款小游戏。其一，黑球发射子弹，躲避红球。其二，黑球平台跳跃。

二、设计

游戏逻辑设计与游戏完整流程

第一个游戏 Barrage：WASD 控制小球，传递给小球加速度，J 键发射子弹，击中敌人会有很大概率将其消灭，碰撞到敌人则会减少生命，屏幕上会有几率出现可以拾取的生命和能量。拾取生命可以回复生命，拾取能量三次可以自动触发大招。血量降到 0 则游戏结束。

第二个游戏 Babel：AD 或者左右键控制小球的移动，小球落在横线上自动反弹，玩家要通过准确地落在横线上不断向上。屏幕视角会强制上升，落出屏幕则游戏结束。

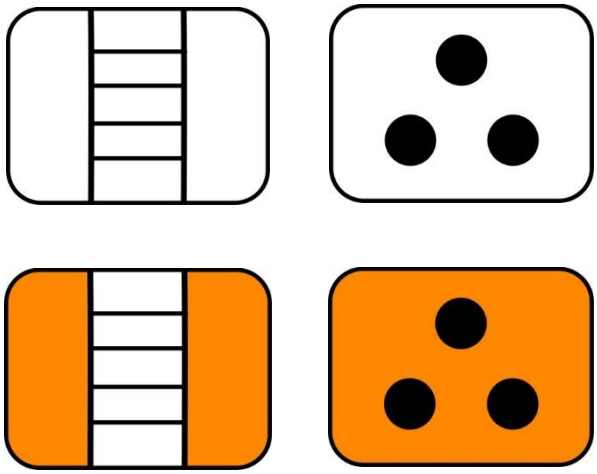
视觉设计

UI 设计

出于美观以及 UI 统一性考虑，不使用 windows 消息或者菜单，制作统一的 UI。

由于是小游戏，UI 设计追求简洁，符号化和抽象化。

左图三点表示第一个弹球游戏 Barrage。梯子表述第二个游戏 Babel。橙色表示选择了该图标，白色表示未选择。



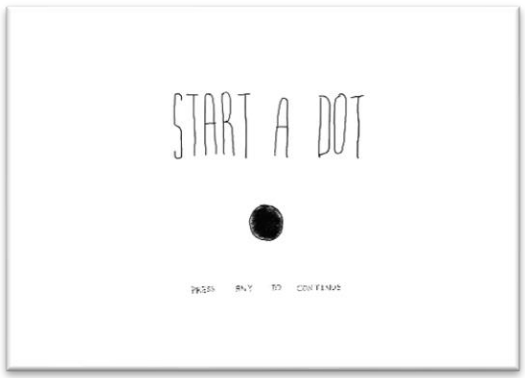
字体设计与选用

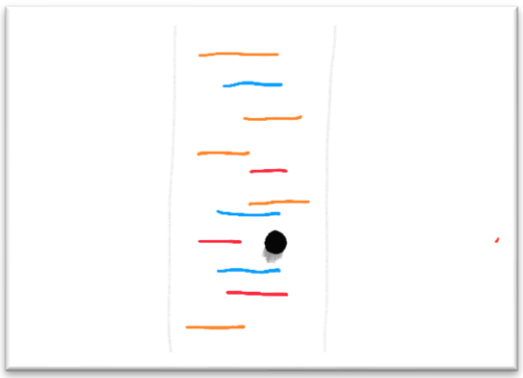
两个标题 START A DOT 和 GAME OVER 由自己设计并绘制矢量图，使用较长的无衬线字体，表达出活泼自由的感受。

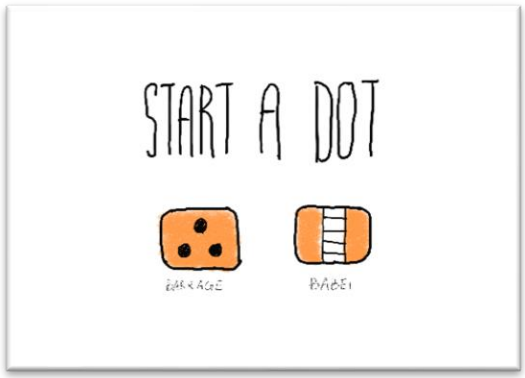
press any key to continue

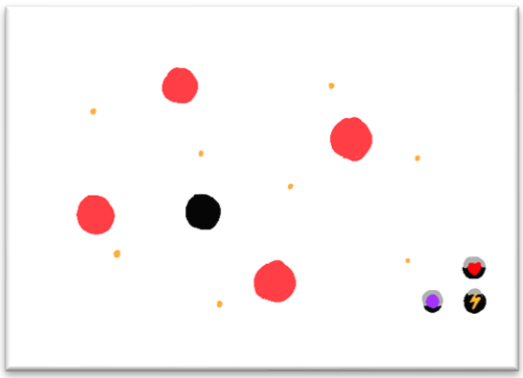
对于 press any key to continue 和 Your score is 100 行字体分别使用 Apple Symbols 和 Verdana 这两个都是比较方的无衬线字体，和标题字体形成对比。

原稿设计





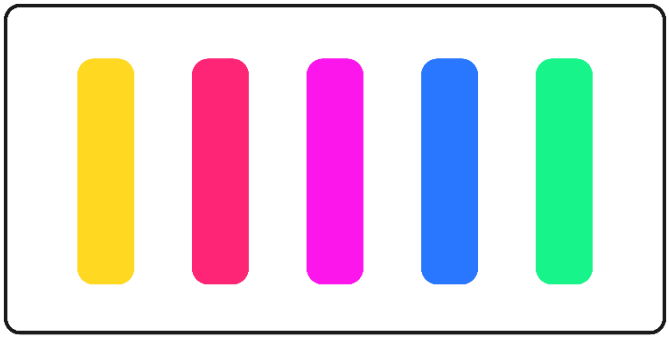




色系的选择

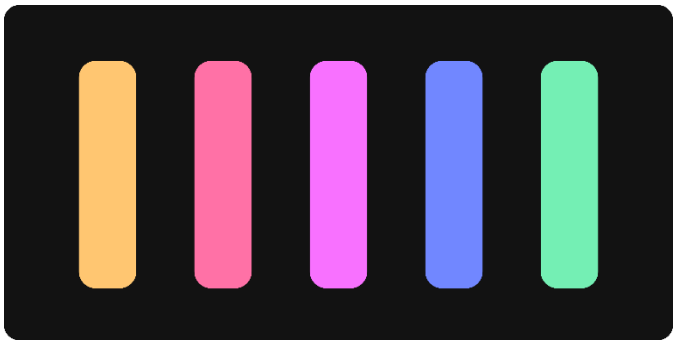
由于是小游戏，选用比较明快活泼的多彩颜色作为搭配。为体现多样性，一个游戏使用亮色配色，另一个使用暗色。

五种颜色在色相环上均匀分布，在亮色系中五种颜色使用 100%的纯度，鲜艳活泼。在暗色系中五种颜色中纯度相应降低



亮色系使用纯度较高的颜色

亮色系配色：底色#FFFFFF 黄色#FFD821 红色#FF2576
紫色#FC16EC 蓝色#2976FF 绿色#16F48A

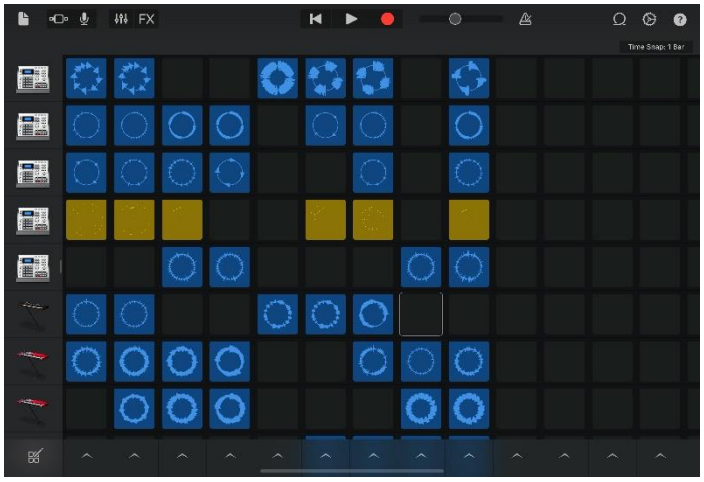


暗色系相应的降低纯度，并且使用 #121212 而非纯黑为底色，适当降低对比度。

暗色系配色：底色#121212 黄色#FFC671 红色#FF71A6
紫色#F871FF 蓝色#7187FF 绿色#74EFB4

音乐设计

灵感来源于实时循环乐段，使用了 Garageband 的官方实时循环乐段，进行了音轨分离然后放入不同部分。



实时循环乐段编辑界面

所有音乐都来自同一个实时循环乐段，拆分出不同情绪的音轨给不同的游戏部分。基本的全音轨版本是 Simple House (feat.Garageband)，其余为拆分版本

交互设计

使用纯键盘交互，第一个带有射击的游戏使用传统的 WASD+JKL 键位，第二个游戏同时支持上下左右键位和 WASD 键位。

三、软件工程

编程 API 选择

性能考虑，由于 Easy X 和 MFC 性能欠缺，GDI 和 GDI+不能很好地对几何图形进行抗锯齿处理，又考虑到现代计算机普遍搭载高性能显卡，所以使用了 Win32 程序搭配 Direct X 中的 Direct 2D 的 2D 图形显示技术以及 Direct Write 图像显示技术，适应 WIC 加载带有透明通道的 PNG 图片，以达到游戏流畅运行图形和动画地目的。

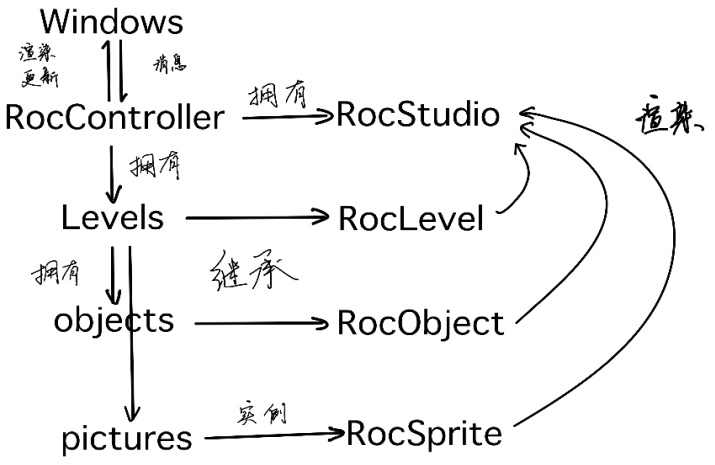
在软件工程方面考虑

1. 考虑对程序进行面向对象封装处理，使得以后对于游戏工程地修改升级能较为方便
2. 保持匈牙利命名法。
3. 完整的异常处理，并且使用 MessageBox 进行反馈
4. 统一使用 Windows 定义数据类型，如 INT 而不是 int。

游戏引擎架构

由于 Direct X 比较复杂，因此首先我编写了一个游戏引擎（取名 ROC 引擎）进行资源管理。

游戏引擎整体架构



游戏引擎整体架构示意图

ROC Controller

ROC 引擎的总控制，拥有 RocStudio 资源，当前关卡指针。并且处理键盘输入，接管合并 WASD 和↑←↓→，并继续传递其他键盘指令给关卡处理。所有的绘画都应由 RocController 进行而非直接调用 RocStudio。所有的关卡加载与卸载也应该由 RocController 进行。

```
class RocController
{
public:
    static VOID Init(HWND hWnd);                        // 初始化

    static VOID LoadInitalLevel(RocLevel* pLevel);    // 加载第一个关卡
    static VOID SwitchLevel(RocLevel* pLevel);        // 卸载且加载关卡

    static VOID BeginDraw();  // 开始 Studio/RenderTarget 绘画资源
    static VOID EndDraw();    // 结束 Studio/RenderTarget 绘画资源
    static VOID Render();     // 调用关卡绘画
    static VOID Update();     // 调用关卡更新

    // 从 WindowProc 处理键盘鼠标输入信息
    static VOID HandleKey(WPARAM wParam, int keyStatus);

    // 判断是否是按下上，接管 'w', 'W', '↑'
    static BOOL IsUp();
    static BOOL IsDown();
    static BOOL IsLeft();
    static BOOL IsRight();

    static RocStudio* GetStudio();

private:
    RocController() { }
    static RocStudio* m_pStudio;          // Studio 指针
    static RocLevel* m_pCurrentLevel;     // 当前关卡指针
    static BOOL m_isLoading;              // 是否加载指示

    // 判断是否是按下上，接管 'w', 'W', '↑'
    static BOOL m_isUp;
    static BOOL m_isDown;
    static BOOL m_isLeft;
    static BOOL m_isRight;

    // 由于 Windows 键盘输入的特殊性，需要记录之前的按键来辅助判断
```

```
    static BOOL m_lastKeyUp;
    static BOOL m_lastKeyDown;
    static BOOL m_lastKeyLeft;
    static BOOL m_lastKeyRight;
};
```

ROC Studio

ROC 引擎的绘画模块，连接 windows 的 window handle。所有资源都调用 RocStudio 进行图像的绘制和渲染。使用 DirectX 模块中的 Direct 2D 绘制图形和图片，Direct Write 绘制 TrueType 文字。

```
class RocStudio
{
public:
    RocStudio();
    ~RocStudio();

    BOOL Init(HWND hWnd);  // 初始化 Studio，申请所有的绘制资源

    VOID BeginDraw();      // 开始绘画
    VOID EndDraw();        // 结束绘画
    ID2D1HwndRenderTarget* GetRenderTarget();  // 获得 RenderTarget

    // 具体绘制方法
    VOID ClearTheScreen(FLOAT r, FLOAT g, FLOAT b);  // 清屏
    VOID DrawText(WCHAR* string, UINT stringLength, D2D1_RECT_F* layout
Rect);  // 绘制文字
    VOID DrawLine(FLOAT x1, FLOAT y1, FLOAT x2, FLOAT y2, FLOAT r, FLOA
T g, FLOAT b, FLOAT a, FLOAT stokeWidth);  // 绘制线
    VOID DrawLine(FLOAT x1, FLOAT y1, FLOAT x2, FLOAT y2, FLOAT rgb, FL
OAT a, FLOAT strokeWidth);
    VOID FillCircle(FLOAT r, FLOAT g, FLOAT b, FLOAT a, FLOAT x, FLOAT
y, FLOAT radius);                          // 绘制圆
    VOID FillCircle(FLOAT rgb, FLOAT a, FLOAT x, FLOAT y, FLOAT radius)
;
    VOID FillSquare(FLOAT r, FLOAT g, FLOAT b, FLOAT a, FLOAT x, FLOAT
y, FLOAT width);                           // 绘制方形
    VOID FillRectangle(FLOAT r, FLOAT g, FLOAT b, FLOAT a, FLOAT x, FLO
AT y, FLOAT width, FLOAT height);          // 绘制矩形

private:
    // 所有的绘制资源
    ID2D1Factory* m_pFactory = NULL;
```

```
    ID2D1HwndRenderTarget* m_pRenderTarget = NULL;
    ID2D1SolidColorBrush* m_pBrush = NULL;
    IDWriteFactory* m_pWriteFactory = NULL;
    IDWriteTextFormat* m_pTextFormat = NULL;
};
```

ROC Level

ROC 引擎的关卡接口，纯虚类，具体的关卡需要继承 RocLevel 并实现相关功能，由 RocController 对关卡进行管理。

```
class RocLevel
{
public:
    virtual VOID Load() = 0;
    virtual VOID Unload() = 0;
    virtual VOID Render() = 0;
    virtual VOID Update() = 0;
    virtual VOID HandleKey(WPARAM wParam) = 0;
};
```

ROC Object

游戏对象的模板，定义并且实现了一些常用的游戏对象的参数和方法，如坐标，速度，销毁。具体的游戏对象的子类由 RocLevel 管理。推荐使用 std::vector 对需要大量生成和销毁的对象进行内存管理

```
class RocObject
{
public:
    VOID  Update();
    virtual VOID Render() = 0;
    virtual BOOL ShouldDestroy() = 0;

    VOID Destroy();

    FLOAT GetX();
    FLOAT GetY();
    FLOAT GetVX();
    FLOAT GetVY();
    VOID  ChangeCoord(FLOAT x, FLOAT y);
    VOID  ChangeCoordX(FLOAT x);
```

```
    VOID  ChangeCoordY(FLOAT y);
    VOID  ChangeSpeed(FLOAT vx, FLOAT vy);
    VOID  ChangeSpeedX(FLOAT vx);
    VOID  ChangeSpeedY(FLOAT vy);

protected:
    FLOAT x;
    FLOAT y;
    BOOL  isStatic;
    FLOAT xSpd;
    FLOAT ySpd;
    BOOL mustDestroy;
};
```

ROC Sprite

ROC 中存储图片的对象，使用 windows code cs (WIC) 读取图片，转化成 direct 2d 的相应格式且存储。

```
class RocSprite
{
public:
    RocSprite(PCWSTR filename, INT width, INT height);
    ~RocSprite();

    VOID Draw(FLOAT x, FLOAT y, FLOAT alpha);

private:
    ID2D1Bitmap* m_pBmp;
};
```

无限生成的数据结构

许多 RocObject 是需要无限生成的，因此使用 std::vector<RocObject*> 来进行比较方便的内存管理：
其中一例：

```
    for (std::vector<Board*>::iterator i = boards.begin(); i != boards.
end();)
    {
        if ((*i)->GetHeight() < m_height - 400)
```

```
        {
            (*i)->Destroy();
            delete (*i);
            i = boards.erase(i);
        }
        else
        {
            ++i;
        }
    }
```

一些具体算法

球与直线的碰撞反弹

直接改变小球相应的 xSpeed 或 ySpeed，且可以进行速度损失，如：xSpd = -xSpd * 0.6;

球与球的反弹

不考虑质量问题，对非对心碰撞进行近似计算：首先计算碰撞的弹力的向量方向，并且给出近似大小，近似大小由小球速度进行近似计算。接着将反弹力的向量和小球原有速度向量相加得到近似的反弹后速度向量。接下来归一化此向量，两小球交换速度，亦可加入一些随机值。

伪代码实现：

```
dx = x1 – x2;
dy = y1 – y1;
newVx1 = vx1 * arg1 + dx * arg2;
newVy1 = vy1 * arg1 + dy * arg2;
newVx2 = vx2 * arg1 - dx * arg2;
newVy2 = vy2 * arg1 - dy * arg2;
```

这种算法并不是严谨的物理计算，但是可以近似的模拟出结果，并且开销较小。

难度递增算法

第一个游戏的难度考量是根据游戏成绩，建立一个多项式函数衡量难度递增，并且设置上限和下限。

例：max_enemy = min(30, score / 10 + 5) // 下限 5 上限 30

```
    for (INT i = 0; i < max_enemy - enemies.size(); ++i) { }
```

第二个游戏的难度考量是根据已游戏时间，建立一个多项式函数衡量难度递增，并且设置上限和下限。