문제해결 및 실습: C++ 과제 보고서

과제 4&5: MFC Final Homework

전공	소프트웨어학과
학번	21011778
이름	염지환
제출일	2024.06.23

세종대학교
SEJONG UNIVERSITY

[목차]

내용

1. 프로그램 구상

I. 클래스 구상

이 프로그램은 다양한 도형들을 그리고, 관리하는 프로그램이므로 각각의 도형마다 기능을 따로 구현하고 화면에 표시하기에는 비효율적이다. 과제 설명에서는 각 도형들을 CMyShape 클래스를 만들어 상속하여 구현하라고 되어있었으므로 CMyShape 클래스를 이용해 도형들을 한꺼번에 편하게 관리할 수 있을 것이라고 생각했다. 따라서 업캐스팅과, 가상함수의 특징을 이용해 효율적으로 프로그램을 만들어 보기로 했다. 우선 CMyShape 클래스로부터 상속하여 CMyRect, CMyCircle 등의 도형 클래스를 만들고, 이를 CMyShape 포인터 리스트에 업캐스팅하여 저장하면 4 개의 도형을 하나의 리스트로 관리할 수 있다. 그리고 이 리스트에 저장된 도형을 각각에 맞게 사용하기 위해서는 공통적으로 쓰이는 함수들의 재정의가 필요하다. CMyShape 에서 가상함수로 선언된 함수를 자식 클래스에서 재정의하면 이 함수는 런타임 시간에 어떤 재정의된 함수를 사용할 지 결정되므로, CMyShape 포인터 리스트를 통해 각 도형에 대한 함수를 호출할 때 각 도형의 클래스에 맞는 함수를 따로 경우의 수를 나누어 호출하지 않아도, 자동으로 클래스에 알맞은 함수를 호출할 수 있다. 이를 잘 이용하면 그리기는 물론 그룹화와 그룹 해제까지 구현할 수 있을 것이라 생각했다.

II. ChildView 기능 구상

이 프로그램을 원활하게 동작하게 하기 위해 구현해야할 기능은 첫째로, 클래스 구현이다. CMyShape 클래스로부터 상속하여 만든 여러 도형 클래스들이 각각 다른 방식으로 그려지기 때문에 이를 연산자 재정의를 이용해 각각 맞추어 주어야 한다. 도형 클래스를 만들었다면, 그 다음은 그림을 그려야 한다. 그림은 마우스 콜백을 통해 그리게 할 수 있으며 CMyShape 포인터 리스트를 통해 그렸던 그림을 저장하여 계속 화면에 유지시킬 수 있다. 이를 위해 도형마다 그리는 방식을 정의해주어야 하며, 특히 선이나 별 같은 경우는 디바이스

컨텍스트 자체에서 제공하는 그리기 함수가 없으므로 이를 생각해서 직접 구현해주어야 한다. 다음으로는 선택과 이동인데, 이 프로그램은 다중 선택과, 그 상태에서 이동하는 기능까지 구현해야한다. 이를 위해 선택된 도형들의 포지션을 저장하는 리스트를 하나 선언하여 선택된 도형들만 따로 관리하는 방법을 택하기로 하였다. 선택 시 나타나는 박스의 경우 CMyShape 에 CRect 형 멤버변수로 선언하여 각 도형마다 하나씩 박스를 저장하고 있을 수 있으면 효율적일 것이라 생각했다. 그리고 도형의 클릭을 판단할 방법 역시 찾아야 했다. 도형의 삭제 같은 경우는 키보드 입력을 통해 선택된 도형들을 전부 지우면 해결되므로 매우 간단하게 구현할 수 있을 것이라 생각했다. 다음으로 가장 어려울 것이라 예상되는 그룹화 및 정렬의 경우 그룹화는 CMyShape 에 자손을 저장할 수 있는 CMyShape 포인터 리스트를 멤버변수로 하나 선언하여 묶인 도형들을 CMyShape 객체 하나로 관리하며 CMyShape 의 함수가 그 자식의 함수를 자동으로 실행시키도록 정의하면 될 것이라 생각했다. 또한 정렬의 경우 도형을 리스트에서 꺼내 리스트의 맨 앞이나 뒤로 삽입하면 해결되는 문제이다. 대략적인 계획을 위화 같이 구상한 채로 프로그램 구현을 시작하였다.

2. 프로그램 구현

I. 클래스 구현

```
// CMyShape 클래스 선언
class CMyShape
{
private:
    CPoint m_pos; // 도형의 위치
    COLORREF m_color; // 도형의 색상
    CList <CMyShape*> m_child; // 자손으로 가지고 있는 도형
protected:
    CRect m_box; // 선택 표시 박스
public:
    CMyShape();
    CMyShape(CPoint pos, COLORREF color);
    CMyShape(CList <CMyShape*>& Selected);
    // draw, drawBox, move, Is_in은 가상함수로 선언하여 재정의하여 사용함
    virtual void draw(CDC& drawDC);
    virtual void drawBox(CDC& drawDC);
    virtual void move(int dx, int dy);
    virtual bool Is_in(CPoint p);
    bool Is_in_box(CRect rect);
    bool hasChild();
    void setPos(CPoint pos);
    CMyShape* getChild();
    CPoint getPos();
    COLORREF getColor();
};
```

우선 가장 중요한 클래스인 CMyShape 함수를 선언하였다. 도형의 위치, 색상, 자손으로 가지는 도형 리스트, 그리고 선택 표시 박스를 멤버 변수로 가지고 있으며, 멤버 함수로는 기본 생성자, 위치와 색상을 받는 생성자, 자손 리스트를 받는 생성자, 도형이 드래그 선택 범위 안에 있는 지 판별하는 Is_in_box 함수, 자손을 가진 객체인지 확인하는 함수 hasChild, 위치를 설정하는 함수 setPos, 자손을 반환하는 함수 getChild, 위치를 반환하는 함수 getPos, 색상을 반환하는 함수 getColor 등이 이 있다. 무엇보다 중요한 함수들은 가상함수로 선언된 4 개의 함수로, draw 는 객체를 화면에 그리는 함수, drawBox 는 선택 표시 박스를 그리는 함수, move 는 객체를 이동시키는 함수, Is_in 은 포인트가 도형 안에 존재하는 포인트인지 확인하는 함수이다. 이 4 가지 함수는 공통적으로 많이 사용하므로 가상함수로 만들어 재정의를 통해 편리하게 사용할 수 있었다.

```
// CMyRect 클래스 선언
class CMyRect : public CMyShape
{
private:
    int m_w, m_h;
public:
    CMyRect(CPoint pos, COLORREF color, int width, int height);
    // draw, drawBox, move, Is_in을 재정의하여 사용함
    virtual void draw(CDC& drawDC);
    virtual void drawBox(CDC& drawDC);
    virtual void move(int dx, int dy);
    virtual bool Is_in(CPoint p);
};
```

```
// CMyCircle 클래스 선언
class CMyCircle : public CMyShape
{
private:
    int m_r;
public:
    CMyCircle(CPoint pos, COLORREF color, int radius);
    // draw, drawBox, move, Is_in을 재정의하여 사용함
    virtual void draw(CDC& drawDC);
    virtual void drawBox(CDC& drawDC);
    virtual void move(int dx, int dy);
    virtual bool Is_in(CPoint p);
};
```

```
// CMyLine 클래스 선언
class CMyLine : public CMyShape
{
private:
    CList <CPoint> m_pts;
public:
    CMyLine(CPoint pos, COLORREF color, CList<CPoint>& pts);
    // draw, drawBox, move, Is_in을 재정의하여 사용함
    virtual void draw(CDC& drawDC);
    virtual void drawBox(CDC& drawDC);
    virtual void move(int dx, int dy);
    virtual bool Is_in(CPoint p);
};
```

```
// CMyStar 클래스 선언
class CMyStar : public CMyShape
{
private:
    CPoint m_pts[10];
public:
    CMyStar(CPoint pos, COLORREF color, CPoint* pts);
    // draw, drawBox, move, Is_in을 재정의하여 사용함
    virtual void draw(CDC& drawDC);
    virtual void drawBox(CDC& drawDC);
    virtual void move(int dx, int dy);
    virtual bool Is_in(CPoint p);
};
```

CMyShape 를 선언한 뒤 CMyShape 를 상속받는 클래스인 CMyRect, CMyCircle, CMyLine, CMyStar 를 선언하였다, 각각은 추가적인 멤버변수로 각각 너비와 높이, 반지름, 포인트 리스트, 크기가 10 인 포인터 배열을 갖는다. 이는 각각 그림을 그리는 데 중요하게 사용되는 변수들이다. 또한 CMyShape 에서 가상함수로 선언했던 함수들이 전부 재정의 되어있다. 이를 통해 ChildView 에서 CMyShape 포인터 리스트 하나로 쉽게 도형들을 관리할 수 있는 틀이 마련되었다.

II. 그리기 구현

클래스를 구현한 뒤에는 ChildView 에서 그림을 그려야 했다. 하지만 우선은 그 이전에 각 도형들을 어떻게 그릴지에 대해 미리 클래스 정의에서 정해두지 않으면 안됐으므로 각 도형마다 그리기 함수를 정의해주었다.

```cpp
// 드로우 컨텍스트를 받아 사각형을 그리는 함수, CMyShape::draw함수의 재정의이다
void CMyRect::draw(CDC& drawDC)
{
    CPoint p = getPos();
    CBrush brush(getColor());
    CPen pen(PS_SOLID, 3, RGB(0, 0, 0));
    drawDC.SelectObject(&brush);
    drawDC.SelectObject(&pen);
    drawDC.Rectangle(p.x - m_w / 2, p.y - m_h / 2, p.x + m_w / 2, p.y + m_h / 2);
}
```

```cpp
// 드로우 컨텍스트를 받아 원을 그리는 함수, CMyShape::draw함수의 재정의이다
void CMyCircle::draw(CDC& drawDC)
{
    CPoint p = getPos();
    CBrush brush(getColor());
    CPen pen(PS_SOLID, 3, RGB(0, 0, 0));
    drawDC.SelectObject(&brush);
    drawDC.SelectObject(&pen);
    drawDC.Ellipse(p.x - m_r, p.y - m_r, p.x + m_r, p.y + m_r);
}
```

```cpp
// 드로우 컨텍스트를 받아 곡선을 그리는 함수, CMyShape::draw함수의 재정의이다
void CMyLine::draw(CDC& drawDC)
{
    CPen pen(PS_SOLID, 3, getColor());
    drawDC.SelectObject(&pen);
    POSITION POS1 = m_pts.GetHeadPosition();
    m_pts.GetNext(POS1);
    POSITION POS2 = m_pts.GetHeadPosition();
    while (POS1 != NULL)
    {
        drawDC.MoveTo(m_pts.GetNext(POS2));
        drawDC.LineTo(m_pts.GetNext(POS1));
    }
}
```

```
// 드로우 컨텍스트를 받아 별을 그리는 함수, CMyShape::draw함수의 재정의이다
void CMyStar::draw(CDC& drawDC)
{
    CPoint p = getPos();
    CBrush brush(getColor());
    CBrush black(RGB(0, 0, 0));
    drawDC.SelectObject(&brush);
    // 크기가 10인 포인트 배열을 이용해 다각형을 만들어 출력한다
    CRgn rgn;
    rgn.CreatePolygonRgn(m_pts, 10, ALTERNATE);
    drawDC.PaintRgn(&rgn);
    drawDC.FrameRgn(&rgn, &black, 3, 3);
}
```

모든 함수들은 ChildView 에서 디바이스 컨텍스트를 매개변수로 받아 호출되어 그림을 그린다. 사각형의 위치는 사각형의 중심으로 정의되어있어, 중심의 좌표와 너비, 높이를 이용해 그림을 그린다. 원의 경우 원의 중심의 좌표와 반지름을 이용하여 원을 그린다. 곡선 같은 경우는 포인트 리스트들에 들어있는 각 점들을 순서대로 MoveTo, LineTo 함수로 이어 비교적 부드러운 곡선을 그리게 하였다. 마지막으로 별 같은 경우 ChildView 에서 별의 10 개의 꼭짓점을 구하는 방법을 구현하여 별의 꼭짓점을 이용해 CRgn 객체를 만들어 윤곽선과 내부를 색칠하는 형식으로 그리도록 만들었다. CRgn 배열을 만들면 편했겠지만 CRgn 객체의 경우 몇 생성자나 연산자 중복이 잘 구현되지 않아 구현을 하다가 막히는 부분이 있었기에, 꼭짓점들을 저장하는 방식으로 구현하였다.

```
public:
    CList <CMyShape*> Shapes; // 도형들을 업캐스팅 하여 저장하는 CMyShape 포인터 리스트
    CList <POSITION> Selected; // 선택된 도형들의 POSITION을 저장하는 리스트
    CPoint p1, p2; // 도형을 그리는데 사용할 점 p1, p2
    CList <CPoint> pts; // 곡선을 그리는데 사용할 포인트 리스트 pts
    CPoint star[10]; // 별을 그리는데 사용할 포인트 배열 star
    COLORREF color; // 현재 색상을 저장할 변수 color
    int radius; // 원, 별을 그리는데 사용할 반지름 radius
    bool dragSelect; // 드래그 선택 모드 여부를 나타내는 dragSelect 변수
    enum Mode { RECT, CIRCLE, CURVE, STAR, SELECT }; // 모드들을 나타내는 열거형 Mode
    Mode mode; // 현재 모드를 저장할 mode 변수
```

다음으로는 ChildView 에서 기능들을 구현하였고 이를 위해 ChildView 헤더파일에 몇 가지 변수들을 선언하고 생성자를 통해 초기화하였다. 제일 중요한 변수는 도형들을 저장하는 Shapes 와 선택된 도형들을 저장하는 Selected 리스트이다.

```
// Shapes를 순회하며 도형을 그리고 선택되어있는 도형의 경우 빨간 점선으로 된 사각형을 그림
POSITION POS1 = Shapes.GetHeadPosition();
while (POS1 != NULL)
{
    POSITION CUR_POS = POS1;
    Shapes.GetNext(POS1)->draw(memDC);
    POSITION POS2 = Selected.GetHeadPosition();
    while (POS2 != NULL)
    {
        if (CUR_POS == Selected.GetNext(POS2))
        {
            Shapes.GetAt(CUR_POS)->drawBox(memDC);
            break;
        }
    }
}
```

그 후 ChildView 에서 그리기를 구현하였다. 화면 깜빡임 방지를 위해 더블 버퍼링을 구현한 뒤, CMyShape 포인터 리스트인 Shapes 와 포지션 변수들을 이용해 Shapes 를 순회하면서 객체를 그리도록 하였다. 여기서 보면 도형의 종류에 상관없이 Shapes.GetNext(POS1)->draw(memDC); 구문 하나로 모든 도형을 그릴 수 있는데, 이는 위에서 구상했듯 클래스 업캐스팅과 함수 재정의을 통해 구현한 방법이다.

```
// 모드에 따라 현재 그리고 있는 도형을 그려냄
if (mode == RECT)
{
    CBrush brush(color);
    CPen pen(PS_SOLID, 3, RGB(0, 0, 0));
    memDC.SelectObject(brush);
    memDC.SelectObject(pen);
    memDC.Rectangle(p1.x, p1.y, p2.x, p2.y);
}
if (mode == CIRCLE)
{
    CBrush brush(color);
    CPen pen(PS_SOLID, 3, RGB(0, 0, 0));
    memDC.SelectObject(brush);
    memDC.SelectObject(pen);
    memDC.Ellipse(p1.x - radius, p1.y - radius, p1.x + radius, p1.y + radius);
}
```

```
if (mode == CURVE)
{
    CPen pen(PS_SOLID, 3, color);
    memDC.SelectObject(pen);
    POSITION POS1 = pts.GetHeadPosition();
    POSITION POS2 = pts.GetHeadPosition();
    if (POS1 != NULL)
        pts.GetNext(POS1);
    while (POS1 != NULL)
    {
        memDC.MoveTo(pts.GetNext(POS1));
        memDC.LineTo(pts.GetNext(POS2));
    }
}
if (mode == STAR)
{
    // 10개의 점을 이용해 rgn으로 별의 형태를 만듦
    CBrush brush(color);
    CBrush black(RGB(0, 0, 0));
    CPen pen(PS_SOLID, 3, RGB(0, 0, 0));
    memDC.SelectObject(brush);
    memDC.SelectObject(pen);
    CRgn rgn;
    rgn.CreatePolygonRgn(star, 10, ALTERNATE);
    memDC.PaintRgn(&rgn);
    memDC.FrameRgn(&rgn, &black, 3, 3);
}
```

다음으로는 현재 모드에 따라 마우스가 왼쪽 클릭된 상태로 드래그 되고 있을 때 드래그에 따라 그림을 그리는 코드이다. 기본적인 구성들은 클래스에서 재정의한 draw 함수들과 매우 유사하다. 처음에는 따로 클래스 객체를 만들어서 draw 함수를 이용해 코드의 길이를 줄이려고 했으나 색상 문제 등 처리해야 할 작업들이 많았기에 길게 작성하였다. 이는 추후 기회가 된다면 충분히 줄일 수 있다고 생각한다.

```
// 도형의 개수와 선택된 도형의 개수를 출력하는 부분
CString str1;
str1.Format(_T("도형의 개수 : %d"), Shapes.GetSize());
memDC.TextOutW(0, 0, str1);

CString str2;
str2.Format(_T("선택된 도형의 개수 : %d"), Selected.GetSize());
memDC.TextOutW(0, 20, str2);
```

다음은 도형과, 선택된 도형의 개수를 출력해주는 부분이다. CString 클래스의 Format 과 TextOuwW 함수로 구현하였으며 Shapes 와 Select 의 사이즈에 따라 자동으로 바뀌도록 만들어 두었다.

```
void CChildView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // 모드에 따라 마우스 클릭 시 동작 구분
    if (mode == RECT)
    {
        SetCapture();
        p1 = point;
        p2 = point;
    }
    if (mode == CIRCLE)
    {
        SetCapture();
        p1 = point;
        p2 = point;
    }
    if (mode == CURVE)
    {
        SetCapture();
        pts.AddTail(point);
    }
    if (mode == STAR)
    {
        SetCapture();
        p1 = point;
        p2 = point;
    }
```

다음으로는 마우스 좌클릭 시 실행되는 함수로, 모드에 따라 마우스 클릭 시의 동작을 구분해 두었다. 대부분은 드래그하여 그림을 그리기 위한 p1,과 p2 를 설정하지만 곡선을 그리는 CURVE 모드의 경우 바로 그 자리의 포인트를 리스트에 추가하는 것을 볼 수 있다.

```
void CChildView::OnMouseMove(UINT nFlags, CPoint point)
{
    // 모드에 따라 좌클릭 + 드래그 시 동작 구분
    if (nFlags & MK_LBUTTON)
    {
        if (mode == RECT)
        {
            p2 = point;
        }
        if (mode == CIRCLE)
        {
            p2 = point;
            // 원의 경우 그리는 과정을 보이기 위해 반지름을 계산
            radius = sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));
        }
        if (mode == CURVE)
        {
            // 선의 경우 점을 계속해서 CPoint 배열에 저장
            pts.AddTail(point);
        }
        if (mode == STAR)
        {
            p2 = point;
            // 별의 경우도 그리는 과정을 보이기 위해 반지름을 계산
            radius = sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));
            // 별의 규칙성을 이용해 10개의 점을 선정하여 추후 rgn 객체의 매개변수로 사용
            for (int i = 0; i < 10; i++)
            {
                float theta = (i * 36 + 90) / (float)180 * 3.141592;
                if (i % 2 == 0)
                    star[i] = CPoint(p1.x - cos(theta) * radius, p1.y - sin(theta) * radius);
                if (i % 2 == 1)
                    star[i] = CPoint(p1.x - cos(theta) * radius/2.6, p1.y - sin(theta) * radius/2.6);
            }
        }
```

다음은 마우스 왼쪽 클릭 후 드래그 시의 동작을 모드에 따라 구분해 두었다. Rect 같은 경우 p2 를 업데이트 하여 사각형을 그리도록 하였고, Circle 은 중앙부터 마우스 까지의 길이를 반지름으로 하는 원을 그리도록 하였다. 선의 경우 계속 점을 포인트 리스트에 넣도록 하였다. 가장 그리기 난해한 도형인 별은 삼각함수를 이용한 좌표의 회전을 이용하여 풀었다. 360 을 10 으로 나누어 i 가 홀수일 경우는 중심부터 마우스까지의 거리와 같은 길이로, 짝수일 경우는 반지름을 2.6 으로 나눈 길이로 0 부터 36 도씩 더해가며 점을 찍었더니 이것으로 별을 그릴 수 있었다. 2.6 같은 경우는 직접 하나하나 실행해가며 찾은 가장 균형이 맞는 별의 짧은 부분의 반지름이었다.

```
void CChildView::OnLButtonUp(UINT nFlags, CPoint point)
{
    // 직사각형의 경우 p1과 p2의 위치관계를 다시 정리하여 CMyRect의 형태로 Shapes에 저
    if (mode == RECT)
    {
        CPoint p, q, center;
        p = p1;
        q = p2;
        if (p.x > q.x)
        {
            q.x = p.x;
            p.x = p2.x;
        }
        if (p.y > q.y)
        {
            q.y = p.y;
            p.y = p2.y;
        }
        center.x = (p.x + q.x) / 2;
        center.y = (p.y + q.y) / 2;
        CMyRect* rect = new CMyRect(center, color, q.x - p.x, q.y - p.y);
        Shapes.AddTail(rect);
        p1 = CPoint(0, 0);
        p2 = CPoint(0, 0);
        ReleaseCapture();
        color = RGB(rand() % 127 + 128, rand() % 127 + 128, rand() % 127 + 128);
    }
    // 원의 경우 원의 중심과 과 반지름을 CMyCircle의 형태로 Shapes에 저장
    if (mode == CIRCLE)
    {
        CMyCircle* circle = new CMyCircle(p1, color, radius);
        Shapes.AddTail(circle);
        p1 = CPoint(0, 0);
        p2 = CPoint(0, 0);
        radius = 0;
        ReleaseCapture();
        color = RGB(rand() % 127 + 128, rand() % 127 + 128, rand() % 127 + 128);
    }
```

```
// 곡선의 경우 선을 이루는 CPoint 리스트를 CMyLine의 형태로 Shapes에 저장
if (mode == CURVE)
{
    CMyLine* L = new CMyLine(CPoint(0,0), color, pts);
    Shapes.AddTail(L);
    pts.RemoveAll();
    ReleaseCapture();
    color = RGB(rand() % 127 + 128, rand() % 127 + 128, rand() % 127 + 128);
}
// 별의 경우 별을 이루는 10개의 꼭짓점 배열을 CMyStar의 형태로 Shapes에 저장
if (mode == STAR)
{
    CMyStar* S = new CMyStar(p1, color, star);
    Shapes.AddTail(S);
    p1 = CPoint(0, 0);
    p2 = CPoint(0, 0);
    radius = 0;
    for (int i = 0; i < 10; i++)
        star[i] = 0;
    ReleaseCapture();
    color = RGB(rand() % 127 + 128, rand() % 127 + 128, rand() % 127 + 128);
}
```

다음은 마우스 좌클릭을 끝내고 손을 땠을 때 실행되는 함수로, 이때 Rect 는 점의 상관관계를 조정한 다음에 CMyRect 를 동적할당하여 저장된다. 다른 도형들 역시 자신을 그리는데 필요한 값들을 매개변수로 하여 자신의 클래스를 동적할당한 뒤, 대입하여 Shapes 배열에 삽입한다. 이렇게 삽입된 도형들은 위에 만들어 둔 함수를 통해 화면에 그려진다. 이를 이용해 화면에 그림을 그리는 것은 성공하였다. 다음은 도형의 선택을 구현할 차례이다.

III. 선택 구현

```
// 선택 모드 시 동작
if (mode == SELECT)
{
    SetCapture();
    p1 = point;
    p2 = point;
    // 도형을 클릭하지 않았을 경우, 드래그 선택모드로 진입
    dragSelect = true;
    // Shift가 눌리지 않았을 시 중복 선택 하지 않음
    if (!(nFlags & MK_SHIFT))
        Selected.RemoveAll();
    POSITION POS = Shapes.GetTailPosition();
    POSITION CUR_POS;
    // 위에 있는 도형을 우선순위로, 선택 리스트에 중복된 도형이 없도록 삽입
    while (POS != NULL)
    {
        CUR_POS = POS;
        if (Shapes.GetPrev(POS)->Is_in(p1) == true)
        {
            POSITION POS2 = Selected.GetHeadPosition();
            bool not_exist = true;
            while(POS2 != NULL)
            {
                if (Selected.GetNext(POS2) == CUR_POS)
                {
                    not_exist = false;
                    break;
                }
            }
            if (not_exist)
                Selected.AddTail(CUR_POS);
            dragSelect = false;
            break;
        }
    }
}
```

이것은 선택 모드에서 마우스 좌클릭 시 실행되는 코드들이다. 먼저 Is_in 함수를 통해 클릭한 장소에 있는 도형이 있는지 확인한다. 우선 도형을 클릭하지 않았을 경우에는 드래그 선택 모드임을 뜻하는 dragSelect 가 true 인 채로 이 함수를 나가게 된다. 그렇지 않고 도형을 클릭했을 경우에는 클릭한 위치에서 제일 앞에 있는 도형이 선택되어 Selected 배열에 저장되게 된다. 또한 도형을 선택하여 저장할 때 반드시 중복 확인을 하는데, 이는 중복 확인을 하지 않았다가 도형 이동시에 중복으로 들어가 있던 도형이 다른 도형보다 심하게 많이 이동하는 등의 문제가 생겨 따로 추가한 코드이다. 그리고 쉬프트를 누르지 않은 상태에서 클릭을 하면 선택된 모든 도형들을 초기화하지만, 쉬프트를 누른 상태로

클릭하면 리스트를 초기화 하지 않도록 설정하여 쉬프트를 누른 상태로 클릭 시 중복 선택이 가능하도록 만들었다. 추가적으로, Is_in 함수는 각 도형의 안에 있거나 또는 곡선에 가까운 곳을 알맞게 클릭해야 판정이 되도록 설정하였는데, 사각형은 범위 안에 있을 경우, 원은 찍은 점과 원의 좌표가 반지름 보다 작을 경우로 판정하였고, 곡선의 경우는 모든 점들의 상하좌우 10px 의 범위에서 클릭이 되도록 판정하였다. 별 같은 경우는 CRgn 객체의 포인트가 포함되어있는지 확인하는 함수인 PtInRegion 함수를 이용하여 클릭 여부를 판정하였다.

```
if (mode == SELECT)
{
    // 드래그 선택 모드인 경우 마우스만 계속 추적
    if(dragSelect)
        p2 = point;
    // 선택 및 이동상태일 경우 p1, p2를 계속 갱신하여 dx, dy를 구해 도형 이동
    else
    {
```

```
// 드래그 선택 상태일 경우 검은색 점선으로 범위를 표시함
if (mode == SELECT && dragSelect)
{
    CPen pen(PS_DOT, 1, RGB(0, 0, 0));
    memDC.SelectStockObject(NULL_BRUSH);
    memDC.SelectObject(pen);
    memDC.Rectangle(p1.x, p1.y, p2.x, p2.y);
}
```

다음은 선택 상태에서 마우스 왼쪽클릭을 유지한 상태로 드래그하는 경우에 발생하는 함수로, 이 경우 드래그 선택 모드인지, 또는 도형의 이동인지가 나뉜다. 그러나 우선은 드래그 선택 모드인 것만 보자면 다른 도형을 그릴 때와 비슷하게 p2 를 업데이트 시켜 Onpaint 에서 선택 범위를 드래그 할 때 마다 업데이트 한다.

```
if (mode == SELECT)
{
    // 드래그 선택 모드일 경우 현재 범위 안에 속한 도형들을 선택 리스트에 삽입
    if (dragSelect)
    {
        CPoint p, q;
        p = p1;
        q = p2;
        if (p.x > q.x)
        {
            q.x = p.x;
            p.x = p2.x;
        }
        if (p.y > q.y)
        {
            q.y = p.y;
            p.y = p2.y;
        }
        CRect SRect = CRect(p, q);
        POSITION POS = Shapes.GetHeadPosition();
        POSITION CUR_POS;
        while (POS != NULL)
        {
            CUR_POS = POS;
            if (Shapes.GetNext(POS)->Is_in_box(SRect) == true)
            {
                Selected.AddTail(CUR_POS);
            }
        }
        dragSelect = false;
    }
    p1 = CPoint(0, 0);
    p2 = CPoint(0, 0);
    ReleaseCapture();
}
```

다음은 드래그 선택 모드인 채로 범위를 드래그하여 클릭을 해제했을 경우 실행되는 함수이다 Rect 와 비슷하게 점의 위치를 정리하여 CRect 형으로 저장하고 이를 각 도형의 Is_In_box 를 통해 각 도형이 드래그 범위 안에 들어왔는지 판정하여 범위 안에 있는 도형들을 Selected 리스트에 삽입한다. 이후 dragSelect 를 false 로 다시 초기화해주고 p1 과 p2 도 역시 0 으로 초기화해준다.

IV. 이동 및 삭제 구현

```
else
{
    p2 = point;
    POSITION POS = Selected.GetHeadPosition();
    int dx = p2.x - p1.x;
    int dy = p2.y - p1.y;
    while (POS != NULL)
    {
        Shapes.GetAt(Selected.GetNext(POS))->move(dx,dy);
    }
    p1 = p2;
}
```

다음은 이동 및 삭제 구현이다. 이동은 선택을 한 상태에서 도형을 클릭한 채로 마우스를 드래그 하면 발생한다. 이 함수에서는 p2 를 업데이트하고 마우스의 각 방향의 이동거리인 dx 와 dy 를 구하여 각 도형의 move 함수를 호출하여 각 도형을 이동시키고, p1 을 p2 로 업데이트 해주어 드래그 이벤트가 일어날 때 마다 dx dy 를 계산하여 이동할 수 있도록 만들었다. 여기서 각 도형들의 move 함수는 다음과 같다.

```
// x와 y의 변화량을 받아 객체와 선택 표시 박스를 이동시키는 함수 CMyShape::move 함수의 재정의이다
void CMyRect::move(int dx, int dy)
{
    CPoint pos = getPos();
    pos.x += dx;
    pos.y += dy;
    setPos(pos);
    m_box.left = pos.x - m_w / 2 - 5;
    m_box.right = pos.x + m_w / 2 + 5;
    m_box.top = pos.y - m_h / 2 - 5;
    m_box.bottom = pos.y + m_h / 2 + 5;
}
```

```
// x와 y의 변화량을 받아 객체와 선택 표시 박스를 이동시키는 함수 CMyShape::move 함수의 재정의이다
void CMyCircle::move(int dx, int dy)
{
    CPoint pos = getPos();
    pos.x += dx;
    pos.y += dy;
    setPos(pos);
    m_box.left = pos.x - m_r - 5;
    m_box.right = pos.x + m_r + 5;
    m_box.top = pos.y - m_r - 5;
    m_box.bottom = pos.y + m_r + 5;
}
```

```
// x와 y의 변화량을 받아 객체와 선택 표시 박스를 이동시키는 함수 CMyShape::move 함수의 재정의이다
void CMyLine::move(int dx, int dy)
{
    CPoint p = m_pts.GetHead();
    p.x += dx;
    p.y += dy;
    m_box = CRect(p.x, p.y, p.x, p.y);
    POSITION POS = m_pts.GetHeadPosition();
    while (POS != NULL)
    {
        CPoint p = m_pts.GetAt(POS);
        p.x += dx;
        p.y += dy;
        if (p.x < m_box.left)
            m_box.left = p.x;
        if (p.x > m_box.right)
            m_box.right = p.x;
        if (p.y < m_box.top)
            m_box.top = p.y;
        if (p.y > m_box.bottom)
            m_box.bottom = p.y;
        m_pts.SetAt(POS, p);
        m_pts.GetNext(POS);
    }
}
```

```
// x와 y의 변화량을 받아 객체와 선택 표시 박스를 이동시키는 함수 CMyShape::move 함수의 재정의이다
void CMyStar::move(int dx, int dy)
{
    for (int i = 0; i < 10; i++)
    {
        m_pts[i].x += dx;
        m_pts[i].y += dy;
    }
    m_box.left = m_pts[8].x - 5;
    m_box.right = m_pts[2].x + 5;
    m_box.top = m_pts[0].y - 5;
    m_box.bottom = m_pts[4].y + 5;
}
```

우선 move 함수들은 도형의 위치를 옮긴다. 사각형이나 원 같은 경우는 그저 위치만 옮기면 끝이 난다. 하지만 곡선이나 별의 경우는 점들이 모여 있는 형태의 클래스이므로, 모든 점들에 dx 와 dy 를 적용해주어 위치를 바꿔주었다. 그리고 공통적으로 m_box 의 위치 역시 변경하여 선택 표시 상자의 위치 또한 도형을 잘 따라오도록 구현하였다. 그리고 move 역시 가상함수를 이용해 재정의한 함수들로, 함수를 도형의 구분에 따라 사용하지 않고 한번에 모든 클래스를 처리할 수 있다.

```
void CChildView::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    // 선택 모드 중 DELETE키가 눌렸을 경우 선택된 도형들을 삭제함
    if (mode == SELECT && nChar == VK_DELETE)
    {
        POSITION POS = Selected.GetHeadPosition();
        while (POS != NULL)
        {
            Shapes.RemoveAt(Selected.GetNext(POS));
        }
        Selected.RemoveAll();
    }
    Invalidate();
    CWnd::OnKeyDown(nChar, nRepCnt, nFlags);
}
```

다음은 삭제의 구현으로, 선택 모드인 상태에서 delete 키가 입력되었을 때 Selected 리스트를 순회하며 리스트에 있는 모든 도형들을 Shapes 에서 삭제하는 것으로 간단하게 구현할 수 있다.

V. 그룹화 및 정렬 구현

```
// 선택된 개체들을 CMyShape 포인터 리스트의 형태로 매개변수로 받는 생성자
CMyShape::CMyShape(CList <CMyShape*>& Selected) : CMyShape()
{
    m_box = Selected.GetHead()->m_box;
    POSITION POS = Selected.GetHeadPosition();
    while (POS != NULL)
    {
        // 선택 표시 박스 영역이 도형들을 전부 포함하도록 재정의한다
        CMyShape* Shape = Selected.GetNext(POS);
        if (Shape->m_box.left < m_box.left)
            m_box.left = Shape->m_box.left;
        if (Shape->m_box.right > m_box.right)
            m_box.right = Shape->m_box.right;
        if (Shape->m_box.top < m_box.top)
            m_box.top = Shape->m_box.top;
        if (Shape->m_box.bottom > m_box.bottom)
            m_box.bottom = Shape->m_box.bottom;
        m_child.AddTail(Shape);
    }
}
```

다음은 그룹화의 구현이다 그룹화를 구현하기 위해서는 CMyShape 함수를 잘 활용해야 했다. 그 첫단계로, 자손들의 리스트를 매개변수로 받는 생성자를 만들었다. 처음에는 리스트를 그대로 받으려 했으나 복사 생성자 문제로 참조로 받아야 한다는 사실을 알게 되었다. 그리하여 리스트의 참조를 받아 이를

순회하며 m_child 에 자손 객체들을 삽입해주었다. 이후 선택 표시 박스가 m_child 에 속한 모든 도형들을 포함하는 크기가 되도록 조정해주었다.

```
// 자손에 속한 도형들을 그리게 하는 함수
void CMyShape::draw(CDC& drawDC)
{
    POSITION POS = m_child.GetHeadPosition();
    while (POS != NULL)
    {
        m_child.GetNext(POS)->draw(drawDC);
    }
}
```

다음으로 중요한 함수는 draw 함수이다 CMyShape 객체의 draw 함수는 m_child 의 모든 도형이 draw 함수를 실행하도록 만든다. 따라서 자동으로 자식의 모든 도형을 출력할 수 있다. 나아가 만약에 이미 그룹화 된 그룹을 다시 묶는 방식으로 여러 번 묶더라도, 상위에 있는 CMyShape 의 draw 함수에 의해 하위의 CMyShape 의 draw 함수가 호출될 것이고, 결국에는 맨 아래의 자손 도형들까지 출력되게 될 것이다.

```
// 자손의 모든 도형들을 움직이고, 선택 표시 박스를 함께 이동시키는 함수
void CMyShape::move(int dx, int dy)
{
    POSITION POS = m_child.GetHeadPosition();
    while (POS != NULL)
    {
        CMyShape* Shape = m_child.GetNext(POS);
        Shape->move(dx, dy);

    }
    m_box.left += dx;
    m_box.right += dx;
    m_box.top += dy;
    m_box.bottom += dy;
}
```

Move 함수 역시 draw 와 마찬가지로 작동하기에 그룹화로 인해 상하관계가 여러 번 쌓이게 되더라도 문제없이 모든 도형을 한꺼번에 이동시킬 수 있다.

```
// 매개변수로 받은 포인트가 자손에 속해있는 도형들 내부에 있는지 판별하는 함수
bool CMyShape::Is_in(CPoint p)
{
    POSITION POS = m_child.GetHeadPosition();
    while (POS != NULL)
    {
        if (m_child.GetNext(POS)->Is_in(p) == true)
            return true;
    }
    return false;
}
```

Is_in 함수 역시 마찬가지로 아무리 그룹화된 도형의 범위가 넓어도 정확히 그룹에 속한 도형 안을 클릭해야만 선택이 되도록 할 수 있게 구현되었다.

```
void CChildView::OnGroup()
{
    // m_child를 가진 CMyShape포인터를 하나 만든 뒤, 선택된 개체들을 m_child에 삽입하고.
    // 선택되었던 개체들을 삭제한 뒤 m_child를 자손으로 가지는 CMyShape 객체를 Shapes에 삽입한다.
    CList <CMyShape*> temp;
    POSITION POS = Selected.GetHeadPosition();
    if (POS == NULL)
        return;
    while (POS != NULL)
    {
        temp.AddTail(Shapes.GetAt(Selected.GetNext(POS)));
    }
    POS = Selected.GetHeadPosition();
    while (POS != NULL)
    {
        Shapes.RemoveAt(Selected.GetNext(POS));
    }
    CMyShape* Group = new CMyShape(temp);
    Shapes.AddTail(Group);
    POS = Shapes.GetTailPosition();
    Selected.RemoveAll();
    Selected.AddTail(POS);
    Invalidate();
}
```

위의 성질을 이용해 만든 그룹화 함수이다. CMyShape 포인터 임시 리스트 temp 에 선택된 도형들을 전부 삽입하고, Shapes 에서 선택되었던 도형들을 지운다음, temp 를 CMyShape 포인터 Group 의 동적할당 시의 매개변수로 준다. 이렇게 만들어진 Group 을 Shapes 에 삽입하여 하나의 도형으로 만들고 Selected 의 모든 원소들을 삭제하면 그룹화가 끝난다. 이렇게 만들어진 그룹은 위에서 만든 함수들에 의해 다른 도형들처럼 잘 작동한다.

```
// 그룹 해제
void CChildView::OnUngroup()
{
    // CMyShape포인터에서 m_child에 저장된 자손들을 전부 꺼내어 Shapes에 삽입하고
    // 그룹 해제된 CMyShape 포인터는 삭제한다.
    POSITION POS = Selected.GetHeadPosition();
    POSITION CUR_POS;
    if (POS == NULL)
        return;
    while (POS != NULL)
    {
        CUR_POS = POS;
        CMyShape* shape = Shapes.GetAt(Selected.GetNext(POS));
        Shapes.RemoveAt(Selected.GetAt(CUR_POS));
        Selected.RemoveAt(CUR_POS);
        if (shape->hasChild() == false)
        {
            Shapes.AddTail(shape);
            Selected.AddTail(Shapes.GetTailPosition());
            break;
        }
        while (shape->hasChild())
        {
            Shapes.AddTail(shape->getChild());
            Selected.AddTail(Shapes.GetTailPosition());
        }
    }
    Invalidate();
}
```

그룹 해제의 경우에는 그룹화의 역방향으로 구현하였다. 먼저 선택된 도형들을 순회한다. 각 원소는 임시 변수 shape 에 저장되며 그 후 Selected 에 저장되어 있던 그 원소의 포지션은 삭제된다. 그 후 만약 원소가 자손을 가지고 있지 않을 경우 그 원소를 바로 다시 Shapes 의 마지막에 삽입하고 그 포지션도 역시 Selected 의 마지막에 삽입한다. 만약 자손을 가지고 있다면 자손이 없어질 때까지 자손을 꺼내어 Shapes 에 삽입하고 그 포지션을 Selected 의 마지막에 삽입하는 것을 반복한다. 이렇게 하면 선택된 도형들의 그룹이 해제되고 해제된 뒤에도 해제된 도형들이 전부 선택된 상태를 유지할 수 있다. 처음에는 자손이 없는 경우를 생각하지 못하여 계속 메모리 오류가 발생했고, 조건문을 하나 넣어주어 해결하였다.

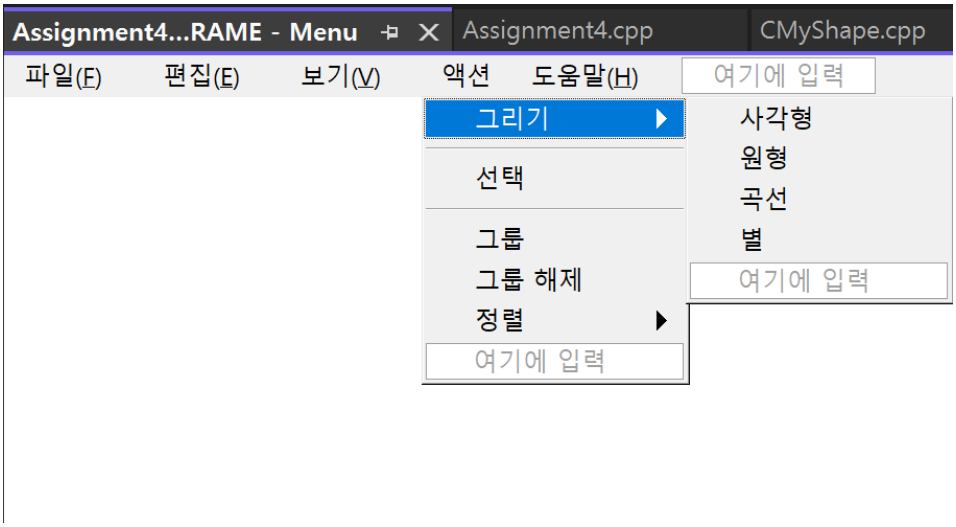
```
void CChildView::OnAlignFront()
{
    // 선택된 도형들을 빼내어 Shapes의 뒤로 다시 삽입한다.
    POSITION POS = Selected.GetHeadPosition();
    POSITION CUR_POS;
    if (POS == NULL)
        return;
    while (POS != NULL)
    {
        CUR_POS = POS;
        Shapes.AddTail(Shapes.GetAt(Selected.GetNext(POS)));
        Selected.AddHead(Shapes.GetTailPosition());
        Shapes.RemoveAt(Selected.GetAt(CUR_POS));
        Selected.RemoveAt(CUR_POS);
    }
    Invalidate();
}
```

다음은 정렬이다. 맨 앞에 원소라는 뜻은 Shapes 에서 가장 늦게 그려지는, 즉 뒤에 있는 원소라는 의미로 선택된 도형들을 빼내어 순서대로 Shapes 의 뒤로 삽입하면 된다. 위의 코드는 while 문을 통해 Selected 를 순회하며 해당하는 원소를 Shapes 의 뒤로 삽입하고 삽입한 원소의 포지션을 Selected 의 앞부분으로 삽입한다. 그리고 Shapes 에 삽입했던 원소를 중간에서 지워버리고 Selected 의 해당 원소 포지션 역시 지워버린다. 처음에는 정렬 후에 두 도형의 앞뒤관계가 뒤바뀌는 경우가 있었는데 원소를 삽입하는 방향을 다시 정하여 이를 해결하였다.

```
void CChildView::OnAlignBack()
{
    // 선택된 도형들을 빼내어 Shapes의 앞으로 다시 삽입한다.
    POSITION POS = Selected.GetTailPosition();
    POSITION CUR_POS;
    if (POS == NULL)
        return;
    while (POS != NULL)
    {
        CUR_POS = POS;
        Shapes.AddHead(Shapes.GetAt(Selected.GetPrev(POS)));
        Selected.AddTail(Shapes.GetHeadPosition());
        Shapes.RemoveAt(Selected.GetAt(CUR_POS));
        Selected.RemoveAt(CUR_POS);
    }
    Invalidate();
}
```

다음은 뒤로 보내는 정렬로, 앞에서 작성했던 앞으로 가져오기 코드랑 정확하게 반대로 작성하면 정상적으로 동작하는 것을 볼 수 있었다.

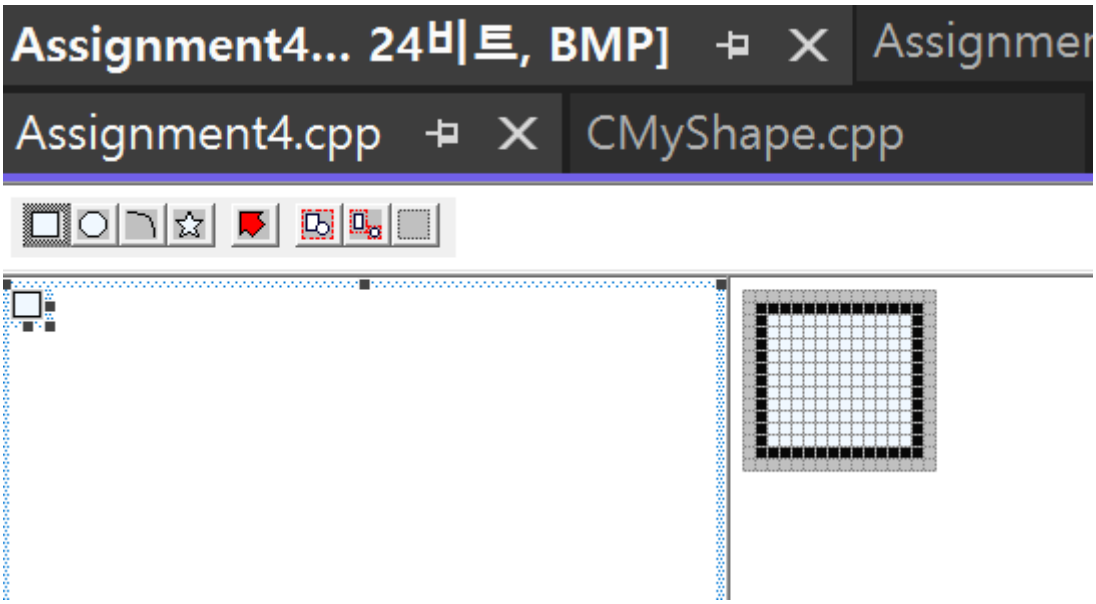
VI. UI 구현



마지막은 UI 구현이다. 메뉴 ID 를 먼저 만들어 놓는 것이 편하기에, 가장 먼저 메뉴를 만들었으며 예제 프로그램과 최대한 동일하게 구분선 또한 작성하였다.

```
// 메뉴 및 툴바 선택에 따라 모드 변경, 그리기 모드 시 선택 리스트를 비움
void CChildView::OnDrawRect()
{
    Selected.RemoveAll();
    mode = RECT;
}

void CChildView::OnDrawCircle()
{
    Selected.RemoveAll();
    mode = CIRCLE;
}

void CChildView::OnDrawCurve()
{
    Selected.RemoveAll();
    mode = CURVE;
}

void CChildView::OnDrawStar()
{
    Selected.RemoveAll();
    mode = STAR;
}

void CChildView::OnSelect()
{
    mode = SELECT;
}
```

그 뒤에는 각 메뉴에 ID 를 할당하여 각 메뉴를 클릭할 때 실행해야 하는 작업을 정의하였다. 여기서는 메뉴 버튼에 따라 모드를 바꿔주었으며 선택모드에서 그리기 모드로 이동할 경우엔 반드시 선택 리스트를 비우도록 구현하였다.



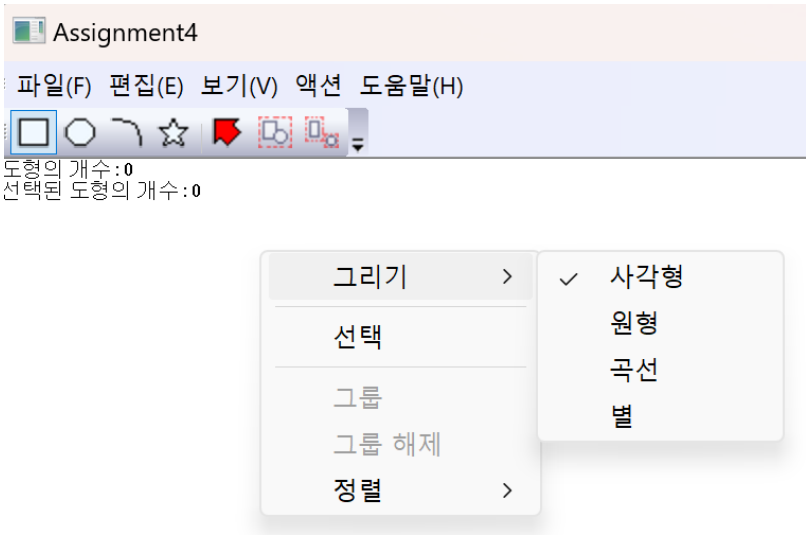
그 뒤로는 툴바를 구현하였고 툴바 버튼에 메뉴에서 할당한 ID 들을 그대로 할당해 주어 메뉴를 클릭했을 때와 같은 역할을 할 수 있도록 만들었다.

```
void CChildView::OnUpdateSelect(CCmdUI* pCmdUI)
{
    if (mode == SELECT)
        pCmdUI->SetCheck(true);
    else
        pCmdUI->SetCheck(false);
}

// 그룹화의 경우 선택된 도형이 2개 이상일 때 활성화
void CChildView::OnUpdateGroup(CCmdUI* pCmdUI)
{
    if (Selected.GetSize() >= 2)
        pCmdUI->Enable(true);
    else
        pCmdUI->Enable(false);
}
```

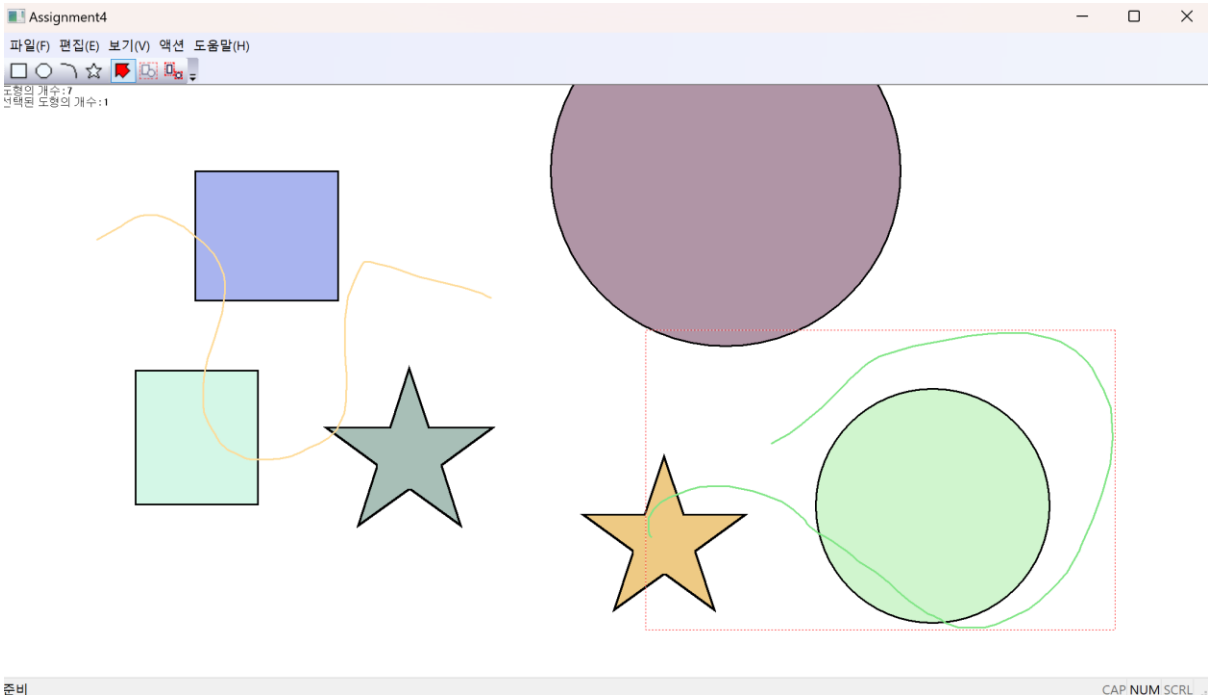
그 다음으로는 툴바와 메뉴의 상태를 업데이트 해주는 함수들을 정의하였다. 이를 통해 툴바에서 사각형 그리기를 선택하면, 사각형 그리기에 파란 불이 들어와 있게 하거나, 선택한 도형이 2 개 이상이어야만 그룹화가 활성화되고, 그룹 해제나 정렬 같은 경우는 선택한 도형이 1 개 이상이어야만 활성화가 되는 등의 작업을 해주었다.

```
// 우클릭 시 콘텍스트 메뉴 열기
void CChildView::OnContextMenu(CWnd* pWnd, CPoint point)
{
    CMenu menu;
    menu.LoadMenu(IDR_MAINFRAME);

    CMenu* pMenu = menu.GetSubMenu(3);

    pMenu->TrackPopupMenu(
        TPM_LEFTALIGN | TPM_RIGHTBUTTON,
        point.x, point.y, AfxGetMainWnd());
}
```

마지막으로 OnContextMenu 를 이용해 우클릭 시에 우클릭 한 위치에 메뉴의 액션과 같은 목록이 나타나도록 설정하였다.



3. 프로그램 완성



프로그램을 완성한 결과 잘 작동하는 것을 볼 수 있었다. 처음에는 객체지향의 성질를 정말 잘 활용하여 효율적이고 간결한 프로그램을 짜보겠다고 결심했지만 점차 코드가 비교적 장황해 지는 것을 보며 객체지향의 원칙을 엄격하게 지키면서 코딩하는 것이 쉬운 일은 아닌 것이라고 체감하게 되었다. 하지만 한편으로는 다형성, 상속과 같은 객체지향의 특성으로 어려워 보이던 문제가 쉽게 해결되는 것을 보고 이를 잘 이용했을 때 얻을 수 있는 이점이 정말 크다는 것도 알게 되었다. 또한 라이브러리를 이용해 간단하지만 짜임새를 갖춘 응용 프로그램을 하나 만들면서 만들어진 도구들을 이용해 무언가를 만드는 데 큰 자신감을 얻을 수 있었다.