

『그래픽 API SkiaSharp을 사용한 광학기기 시뮬레이터 개발』

3학년 의반 13번 박현우

● 초록(Abstract) :

프로그래밍 언어 C#이 속해 있는 .NET 프레임워크의 그래픽 처리 API SkiaSharp은 국내에는 잘 알려져 있지 않아 어느 정도 사용층이 있는 해외에 반해 거의 사용되지 않고 있는 실정이다. 본 탐구에서는 이 SkiaSharp을 사용하여 물리Ⅱ 교과 시간에 학습한 내용을 바탕으로 광학기기 시뮬레이터를 제작하기 위한 연구를 진행하였고, 그 결과 C#과 SkiaSharp을 사용해서 광학기기의 종류와 물체의 위치에 따른 광선의 경로와 물체의 상의 정확한 위치와 모양을 물리학적 관점을 통해 수학적으로 계산하여 사용자에게 시각적으로 보여주는 시뮬레이터를 개발하는데 성공하였다.

● 목 차(Index) :

I. 머리말

1. 탐구 목적 및 문제제기
2. 탐구의 필요성과 연계성
3. 선행연구에 대한 분석 및 연구의 방향 제시

II. 이론적 배경

1. 광학기기와 상
2. SkiaSharp

III. 연구방법 및 주요 내용

1. SkiaSharp 커스텀뷰 설계 및 구현
2. 시뮬레이션 코드 설계 및 구현

IV. 맺음 말

위 사본은 원본과 상이없음을

2018. 09. 18

청 원 고 등 학 교



I. 머리말

1. 탐구 목적 및 문제제기

C#이 속해있는 .NET 프레임워크의 그래픽 처리 API 중 하나인 SkiaSharp은 국내에서는 거의 사용되고 있지 않으며, 당장 네이버에 검색하기만 해도 자료의 부족이 심각해 국내에서 SkiaSharp을 사용해보려는 사람들은 큰 불편함을 겪을 것으로 보인다.

이와 별개로 물리Ⅱ 3단원에 수록되어있는 광학기와 상에 관한 내용은 교과서의 내용만 가지고는 처음 배우는 학생들에게 광학기와 물체의 위치에 따른 상의 특징이 머릿속에 직관적으로 잘 와 닿지 않는 등 학습에 어려움이 있을 것으로 보인다.

이에 SkiaSharp을 사용하여 광학기기에서 광선과 물체의 상의 위치와 모양 등을 계산하여 이를 시각적으로 보여주는 시뮬레이션 소프트웨어를 프로그래밍을 통해 직접 제작해보고자 한다.

2. 탐구의 필요성과 연계성

직접 광학기기 시뮬레이터를 제작하는 과정에서 SkiaSharp을 이용하여 C#에서 그래픽을 처리하는 방법을 조사함으로써 국내에 전무한 SkiaSharp에 대한 정보와 자료를 늘릴 수 있을 것이며, 국내 프로그래머들의 SkiaSharp 개발 활성화에 기여할 수 있을 것으로 기대된다.

동시에 광학기기를 지나는 광선과 물체의 상의 구현에 필요한 수학적, 물리학적 지식을 프로그래밍 언어로 설계 및 구현하는 연구를 거치면서 이러한 지식들, 특히 광학기기에 대한 이해를 더욱 굳건히 할 수 있다. 궁극적으로 결과물인 광학기기 시뮬레이터를 수업 중 학생들의 광학기기 학습에 유용하게 사용할 수 있을 것으로 기대된다.

3. 선행연구에 대한 분석 및 연구의 방향 제시

앞서 설명했듯이 국내에서 SkiaSharp에 관한 자료나 연구는 거의 존재하지 않는다. Xamarin 환경을 중점으로 나쁜 연구가 활발한 해외와 달리 국내에서

는 순수 C#/WPF 환경은 고사하고 Xamarin 환경에서의 SkiaSharp을 사용하는 방법에 관한 자료도 상당히 부족한 편이다.

광학기기 시뮬레이터는 이미 국내 온라인에 존재하지만, 몇 가지 아쉬운 점이 있다. 먼저 거울과 렌즈가 두 소프트웨어에 분리되어있고, 물체의 크기를 조절할 수 없게 되어있다. 또 거울에 오목거울은 2개, 볼록거울은 3개의 광선밖에 표시되어 있지 않고 오목거울의 도립실상이 좌우대칭이 안 되어있으며, 렌즈는 초점을 조절할 수 없게 되어있다 [1][2][3].

따라서 거울과 렌즈를 통합하고, 거울은 4개, 렌즈는 3개의 광선을 표시함과 동시에 물체의 크기와 초점 조절까지 가능하며 상의 모양과 방향도 정확히 표시하는 광학기기 시뮬레이터를 C#/WPF 환경에서 SkiaSharp API를 사용하여 구현하고, 그 과정에서 필요한 SkiaSharp 사용법과 광학기기의 물리학적 특성을 프로그래밍으로 구현하는 것에 관한 연구를 중점적으로 하도록 한다.

II. 이론적 배경

1. 광학기기와 상

가. 광학기기

1) 구면 거울

구면 거울은 구면의 일부를 잘라내서 만든 거울이다. 이 중 반사면이 오목한 거울을 오목거울, 볼록한 거울을 볼록거울이라고 한다 [4].

2) 렌즈

렌즈는 빛의 굴절을 이용하여 상을 만드는 광학기기이다. 가장자리보다 가운데 부분이 두꺼운 렌즈를 볼록렌즈, 가장자리보다 가운데 부분이 얇은 렌즈를 오목렌즈라고 한다 [4].

나. 광학기기에 의한 상

1) 구면 거울에 의한 상

구면 거울에 입사하는 모든 광선은 반사 법칙을 따르며, 물체의 어떤 한 점에서 나와 거울에 입사하는 광선 중 어떠한 특징을 보여 상의

작도에 유용하게 사용되는 광선들이 4가지 정도 존재한다.

첫 번째로, 거울 축 가까이에서 축에 평행하게 입사한 광선은 반사 후 초점이라는 특정한 점을 지나거나(오목거울), 초점에서 나온 것처럼(볼록거울) 반사하는 특징이 있다.

두 번째로, 초점을 향해 입사한 광선은 거울 축에 나란한 방향으로 반사하는 특징이 있다.

세 번째로, 구심을 향해 입사한 광선은 반사 후 그대로 되돌아오는 특징이 있다.

네 번째로, 거울면의 중심으로 입사한 광선은 거울 축에 대칭, 즉 축을 법선으로 하여 반사하는 특징이 있다.

본 탐구에서는 위 4가지 광선을 각각 1번, 2번, 3번, 4번 광선이라고 정의한다.

구면거울에서 물체에서 거울까지의 거리(a), 거울에서 상까지의 거리(b), 거울면의 중심에서 초점까지의 거리(f) 사이의 관계를 나타낸 관계식이 존재하는데, 다음과 같다.

$$\frac{1}{a} + \frac{1}{b} = \frac{1}{f}$$

이 때 b 값은 실상인 경우에 (+), 허상인 경우에 (-)값을 가진다. f 값은 오목거울일 때 (+), 볼록거울일 때 (-)값을 가진다 [4].

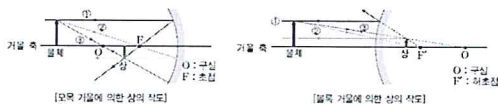


그림 1-1. 구면 거울에 의한 상의 작도

2) 렌즈에 의한 상

렌즈 또한 구면 거울과 마찬가지로 어떠한 특징을 보이는 광선들이 존재하는데, 다음과 같이 3가지 정도 존재한다.

첫 번째로, 렌즈 축에 평행하게 입사한 광선은 초점을 지나거나(볼록렌즈) 초점에서 나온 것처럼(오목렌즈) 굴절한다.

두 번째로, 초점을 지나거나(볼록렌즈) 초점을 향해(오목렌즈) 입사한 광선은 광축에 나란한 방향으로 굴절한다.

세 번째로, 렌즈의 중심으로 입사한 광선은 렌즈를 지난 후 그대로 직진한다.

본 탐구에서는 위 광선을 각각 1번, 2번, 4번 광선이라고 정의한다. 3번이 아닌 4번인 이유는 거울과 렌즈 사이에 유사한 특징을 보이는 광선끼리 분류해 나중에 코딩을 용이하게 하기 위함이다.

얇은 렌즈일 때 물체에서 렌즈까지의 거리(a), 렌즈에서 상까지의 거리(b), 렌즈의 중심에서 초점까지의 거리(f) 사이의 관계를 나타낸 관계식이 존재하는데, 다음과 같다.

$$\frac{1}{a} + \frac{1}{b} = \frac{1}{f}$$

이 때 b 값은 실상인 경우에 (+), 허상인 경우에 (-)값을 가진다. f 값은 볼록렌즈일 때 (+), 오목렌즈일 때 (-)값을 가진다 [4].

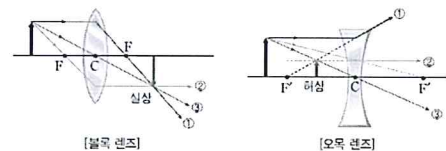


그림 1-2. 렌즈에 의한 상의 작도

2. SkiaSharp

가. 개요

다른 언어가 그렇듯이 C# 역시 언어 그 자체만으로 복잡한 기능을 수행하는 소프트웨어를 개발하는 것에 한계가 있어, 원하는 기능을 보다 쉽고 간편하게 개발할 수 있도록 도와주는 API가 존재한다. SkiaSharp은 C#이 속한 .NET 프레임워크에서 그래픽 처리를 도와주는 API 중 하나이다.

나. 주요 메소드

SKCanvas.Clear: SKCanvas 내부 전체를 투명한(또는 원하는) 색으로 바꾸는 메소드이다. 여기서 SKCanvas는 실제 그래픽 처리의 결과물을 그릴 때 사용되는 클래스로 글자 그대로 미술에서의 캔버스와 유사하다고 할 수 있다.

SKCanvas.DrawLine: 글자 그대로 원하는 위치에 직선을 그려주는 메소드이다. 이것을 포함한

이하 Draw 계열의 메소드는 매개변수로 SKPoint 나 SKRect를 요하는데, 이는 이미지를 그릴 위치 나 영역의 좌표를 저장하는 구조체이다.

SKCanvas.DrawPoints: 원하는 위치에 매개변수에 따라 점 또는 연결된 직선 등을 그려주는 메소드이다. 본 프로젝트에서는 광선 경로를 그릴 때 주로 사용하였다.

SKCanvas.DrawBitmap: 캔버스에 원하는 그림을 그려주는 메소드이다 [5]. 본 프로젝트에서는 광학기기, 물체, 상을 그릴 때 주로 사용하였다.

III. 연구방법 및 주요 내용

1. SkiaSharp 커스텀뷰 설계 및 구현

1) 커스텀뷰 구조 설계

먼저 SkiaSharp으로 그린 결과물을 사용자가 보는 화면에 출력해주는 뷰(View)가 있어야 한다. 물론 SkiaSharp 내부에 자체 뷰가 이미 존재하지만, 이번 개발에 사용할 MVVM 디자인 패턴에서 이를 그대로 사용하는 것에는 약간 무리가 있다. 따라서 MVVM 디자인 패턴을 위한 커스텀뷰를 직접 제작해 사용하고자 한다.

실질적인 코딩에 앞서 구현할 커스텀뷰의 구조를 체계적으로 설계하고자 한다. 이 때 커스텀뷰는 차후 타 프로젝트에 쓰일 때를 대비해 현 프로젝트에 종속되지 않고 별도의 수정 없이 즉각 타 프로젝트에 적용할 수 있도록 최대한 독립적인 구조로 구현하려고 한다. 다음은 위 기준에 부합하는 커스텀뷰의 구조를 설계한 것이다.

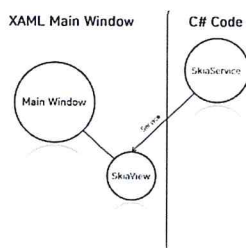


그림 2-1. 개략적인 커스텀뷰 구조

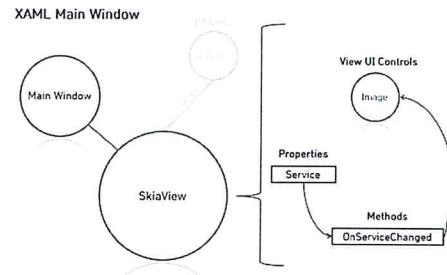


그림 2-2. SkiaView 세부구조

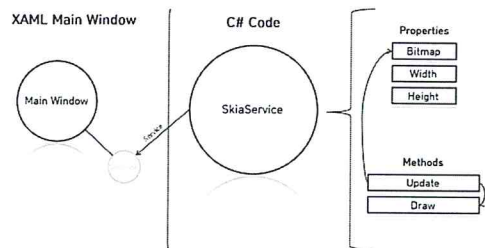


그림 2-3. SkiaService 세부구조

SkiaView라는 이름의 커스텀뷰에는 결과를 화면에 출력하는 이미지 컨트롤과 Service라는 이름의 속성(Property)이 하나씩 존재한다. 이 속성에는 SkiaService라는 형식의 객체를 값으로 설정할 수 있으며, 설정한 객체에 따른 작업 결과가 이미지 컨트롤에 전달되어 화면에 출력되는 구조이다. 여기서 SkiaService 또한 SkiaSharp에는 본래 존재하지 않는 커스텀 클래스로, 프로그래머가 원하는 기능을 수행하는 그래픽 처리를 구현하는 것을 도와줄 추상 클래스로 설계하였다. SkiaService를 상속한 클래스를 새로 만들어 Draw 메소드를 재정의(overriding)해 프로그래머가 원하는 기능을 코딩하고 커스텀뷰의 속성을 이것으로 바꿔주기만 하면 소프트웨어가 화면에 출력할 내용을 한 번에 바꿔 손쉽게 다른 기능을 수행하도록 할 수 있기에, 이는 커스텀뷰가 프로젝트에 독립적이어야 한다는 기준에 충실히 부합한다고 할 수 있다.

2) 커스텀뷰 구현

이제 위에서 설계한 구조에 부합하는 커스텀뷰를 코딩을 통해서 구현하고자 한다. 앞서 설명했듯이 프로그래밍 언어는 C#을 사용하며, 실제 화면에 보이는 UI 부분은 그래픽 하부 시스템

WPF를 이용하여 구현하였다. 다음은 작성한 코드의 일부이다 [6].

```
<Grid>
    <Image x:Name="image"
    Stretch="Fill"/>
</Grid>
```

코드 1-1. SkiaView.xaml

```
namespace OpticalSimulator.View
{
    public partial class SkiaView :
    UserControl
    {
        public static readonly
        DependencyProperty ServiceProperty =
        DependencyProperty.Register("Service",
        typeof(SkiaService), typeof(SkiaView), new
        PropertyMetadata(OnServiceChanged));

        private static void
        OnServiceChanged(DependencyObject d,
        DependencyPropertyChangedEventArgs e)
        {
            SkiaView view = d as
            SkiaView;
            if (view != null)
            {
                SkiaService service =
                e.NewValue as SkiaService;
                view.image.Source =
                service.Bitmap;

                CompositionTarget.Rendering += (obj,
                args) => service.Update();
            }
        }


        public SkiaService Service
        {
            get { return
            (SkiaService)GetValue(ServiceProperty); }
            set { SetValue(ServiceProperty,
            value); }
        }
    }
}
```

```
}
}
}
```

코드 1-2. SkiaView.xaml.cs

```
namespace OpticalSimulator.Service
{
    public abstract class SkiaService
    {
        public WriteableBitmap Bitmap {
        get; private set; }
        public int Width { get; private set; }
        public int Height { get; private set; }

        public SkiaService(int width, int
        height)
        {
            Bitmap = new
            WriteableBitmap(width, height, 96, 96,
            PixelFormats.Bgra32,
            BitmapPalettes.Halftone256Transparent);
            Width = width;
            Height = height;
        }

        public void Update()
        {
            
            (생략)
            Draw(canvas);
            (생략)
        }

        protected abstract void
        Draw(SKCanvas canvas);
    }
}
```

코드 1-3. SkiaService.cs

2. 시뮬레이션 코드 설계 및 구현

1) 광선 표시 코드 설계 및 구현

이제 본격적으로 사용자의 요구에 적합한 광선 경로를 수학적, 물리학적으로 계산해 시각적으로

화면에 출력해주는 기능을 구현하고자 한다. 먼저 계산 및 그 결과를 화면에 보여주는 기능을 담당할 SkiaService의 서브클래스인 OpticalService 클래스를 만들고, Draw 메소드를 재정의한다.

본격적인 코드 작성 전에 먼저 윈도우 좌표계에 대해서 말하고자 한다. 아래와 같이 윈도우 좌표계는 수학에서 사용하는 좌표계와는 달리 아래쪽으로 내려갈수록 y좌표 값이 증가하는 형태인데, 쉽게 말해서 수학의 좌표계에서 x축 대칭을 했다고 보면 된다 [7].

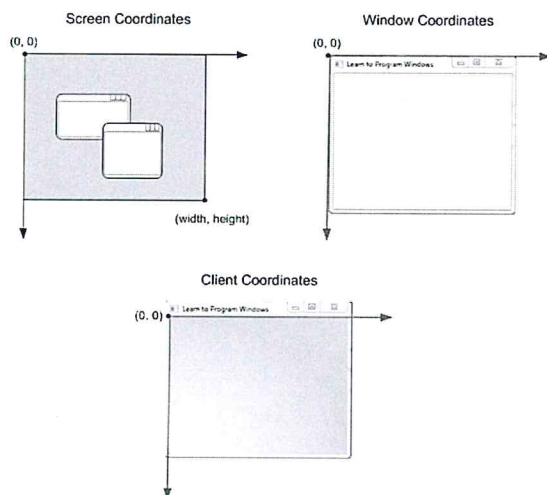


그림 3. 윈도우 좌표계

하지만 여기서 개발하고 있는 시뮬레이터에서는 코딩에 용이하도록 광학기가 위치해있는 정가운데를 원점으로 잡고자 한다. 따라서 다음과 같이 정가운데의 좌표를 originPoint라는 이름의 변수로 선언하고 이를 원하는 좌표 값과 +, 연산자로 합성함으로써 정가운데의 좌표를 원점처럼 사용하여 프로그래밍을 할 수 있게 하였다.

```
// 원점
SKPoint originPoint = new SKPoint(Width / 2, Height / 2);
```

이제 위 변수와 a , f 값을 이용해서 광선의 위치를 표현하고 이를 SkiaSharp을 사용해 코딩하면 물체의 위치와 초점거리에 따른 광선의 입사 및 반사 상태를 시뮬레이션할 수 있게 된다. 다음은 구현한 코드의 일부이다.

```
// 초점, 구심 좌표
SKPoint[] xPoints = new SKPoint[2];
// 물체 좌표
SKPoint objPoint = originPoint + new SKPoint(-A, -ObjectSize);

// 1번 광선 경로
SKPoint[] points_1 = new SKPoint[3];
points_1[0] = objPoint;
points_1[1] = originPoint + new SKPoint(0, -ObjectSize);

// 2번 광선 경로
SKPoint[] points_2 = new SKPoint[3];
points_2[0] = objPoint;

// 3번 광선 경로
SKPoint[] points_3 = new SKPoint[2];

// 4번 광선 경로
SKPoint[] points_4 = new SKPoint[3];
points_4[0] = objPoint;
points_4[1] = originPoint;

// 1번 광선 경로 (가상)
SKPoint[] points_1v = new SKPoint[2];
// 2번 광선 경로 (가상)
SKPoint[] points_2v = new SKPoint[2];
// 3번 광선 경로 (가상)
SKPoint[] points_3v = new SKPoint[2];
// 4번 광선 경로 (가상)
SKPoint[] points_4v = new SKPoint[2];

// 광학기 종류별 처리
switch (Optical)
{
    case OpticalType.oga:
        (생략)
        break;
    case OpticalType.bolga:
        (생략)
        break;
    case OpticalType.bollen:
        (생략)
        break;
}
```



```

        case OpticalType.olen:
            (생략)
            break;
    }

    // 광축 표시
    canvas.DrawLine(0, Height / 2, Width,
        Height / 2, linePaint);
    // 초점 표시
    canvas.DrawPoints(SKPointMode.Points,
        xPoints, pointPaint);

    // 1번 광선 표시
    canvas.DrawPoints(SKPointMode.Polygon,
        points_1, new SKPaint() { StrokeWidth = 2,
        Color = SKColors.Orange });
        (생략)
    // 4번 광선 표시
    canvas.DrawPoints(SKPointMode.Polygon,
        points_4, new SKPaint() { StrokeWidth = 2,
        Color = SKColors.DeepPink });

    // 1번 광선 표시 (가상)
    canvas.DrawPoints(SKPointMode.Lines,
        points_1v, new SKPaint() { StrokeWidth = 2,
        Color = SKColors.Orange, PathEffect =
        SKPathEffect.CreateDash(new float[] { 5f, 5f
        }, 20f) });
        (생략)
    // 4번 광선 표시 (가상)
    canvas.DrawPoints(SKPointMode.Lines,
        points_4v, new SKPaint() { StrokeWidth = 2,
        Color = SKColors.DeepPink, PathEffect =
        SKPathEffect.CreateDash(new float[] { 5f, 5f
        }, 20f) });

    // 광학 기기 표시
    canvas.DrawBitmap(opticalBitmap,
        originPoint + new
        SKPoint(-opticalBitmap.Width / 2,
        -opticalBitmap.Height / 2));

```

코드 2-1. OpticalService.cs -광선-

2) 상 표시 코드 설계 및 구현

앞서 설명하였던 구면거울과 렌즈의 관계식을

이용하여 광학기와 상 사이의 거리를 구한 후, 배율의 크기와 부호를 고려해 적절한 위치에 상을 그리고자 한다. 다음은 구현한 코드의 일부이다.

```

// B값
if (Optical == OpticalType.oga || Optical ==
    OpticalType.bollen)
    B = A * F / (A - F);
else if (Optical == OpticalType.bolga ||
    Optical == OpticalType.olen)
    B = A * -F / (A + F);

// 배율
M = B / A;
// 상 크기
ImageSize = ObjectSize * Math.Abs(M);
        (생략)
if (objectBitmap != null)
{
    // 물체 표시
    canvas.DrawBitmap(objectBitmap,
        SKRect.Create(originPoint + new SKPoint(-A
        - ObjectSize / objectBitmap.Height *
        objectBitmap.Width / 2, -ObjectSize), new
        SKSize(ObjectSize / objectBitmap.Height *
        objectBitmap.Width, ObjectSize));

    // 상 표시
    if (Optical == OpticalType.oga || Optical
        == OpticalType.bolga)
    {
        if (M > 0)
            (생략)
        else if (M < 0)
            (생략)
    }
    else if (Optical == OpticalType.bollen ||
        Optical == OpticalType.olen)
    {
        if (M > 0)
            (생략)
        else if (M < 0)
            (생략)
    }
}

```

코드 2-2. OpticalService.cs -상-

IV. 맺음 말

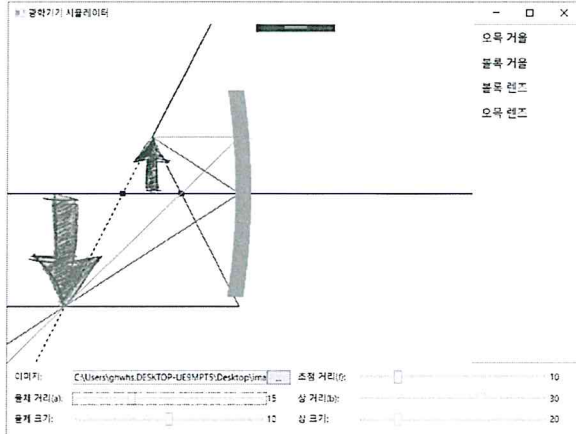


그림 4. 최종적으로 완성한 광학기기 시뮬레이터

SkiaSharp은 구글에서 개발한 Skia라는 그래픽 엔진을 .NET에서 사용할 수 있도록 바인딩한 엔진이면서, 크로스 플랫폼 앱 개발 프레임워크인 Xamarin과의 호환성도 뛰어난 등 여러모로 이점이 많아 해외에서는 어느 정도 쓰이는 API이지만, 유독 국내에서는 관련 정보의 부족이 심한 편이고, 무엇보다 이 API의 존재를 아는 사람도 극히 드문 편이다 [5]. 이번에 SkiaSharp을 사용해본 결과 C#에서 그래픽 처리를 하기에 나름 괜찮은 API인 것 같은데 거의 사용되지 않아 아쉽다고 느꼈다. 그러므로 이번 탐구를 계기로 삼아 앞으로 SkiaSharp에 대한 조사와 연구를 좀 더 진행하고 국내 프로그래밍 커뮤니티에 자료나 정보 등을 주기적으로 공유함으로써 국내 SkiaSharp 개발 활성화에 기여하고자 한다.

또한 광학기기 시뮬레이터를 제작하기 위해 광학기기에 대한 내용을 조사하면서 그 전까지 이해했다고 생각했던 내용 중 사실 막연하게만 알고 있던 내용이 많았다는 것을 깨달았고 이번 탐구를 통해서 광학기기에 대한 지식을 좀 더 엄밀하게 정리할 수 있게 되었고, 또 책에 소개되어있는 광선들을 특징에 맞게 구현하기만 했을 뿐인데 광선이 자연스럽게 전부 한 점에서 모이는 것을 두 눈으로 관찰하는 등 물리학에 대한 여러 색다른 경험을 할 수 있었다.

본 보고서에는 소프트웨어의 소스코드 전부가 아

닌 핵심적인 기능을 담당하는 일부만을 담았다. 대신 깃허브에 프로젝트의 소스코드를 전부 업로드 하였으니, <https://github.com/Winrobrine/OpticalSimulator>에 들어가면 확인할 수 있다.

< 참고 문헌 >

- [1] 윤제한의 물리교실. “오목거울 볼록거울”
<http://yjh-phys.tistory.com/24>.
- [2] 윤제한의 물리교실. “오목렌즈 볼록렌즈”
<http://yjh-phys.tistory.com/26>.
- [3] 윤제한의 물리교실. “오목거울과 볼록거울”
<http://yjh-phys.tistory.com/29>.
- [4] 류상호 외. 완자 물리II.
- [5] Microsoft. “Using SkiaSharp in Xamarin.Forms.”
<https://docs.microsoft.com/en-US/xamarin/xamarin-forms/user-interface/graphics/skiasharp>
- [6] Github. “SkiaSharp-Wpf-Example.”
<https://github.com/8/SkiaSharp-Wpf-Example>
- [7] Microsoft. “What is a Window?”
[https://msdn.microsoft.com/en-us/library/windows/desktop/ff381403\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ff381403(v=vs.85).aspx)

2019.10.10