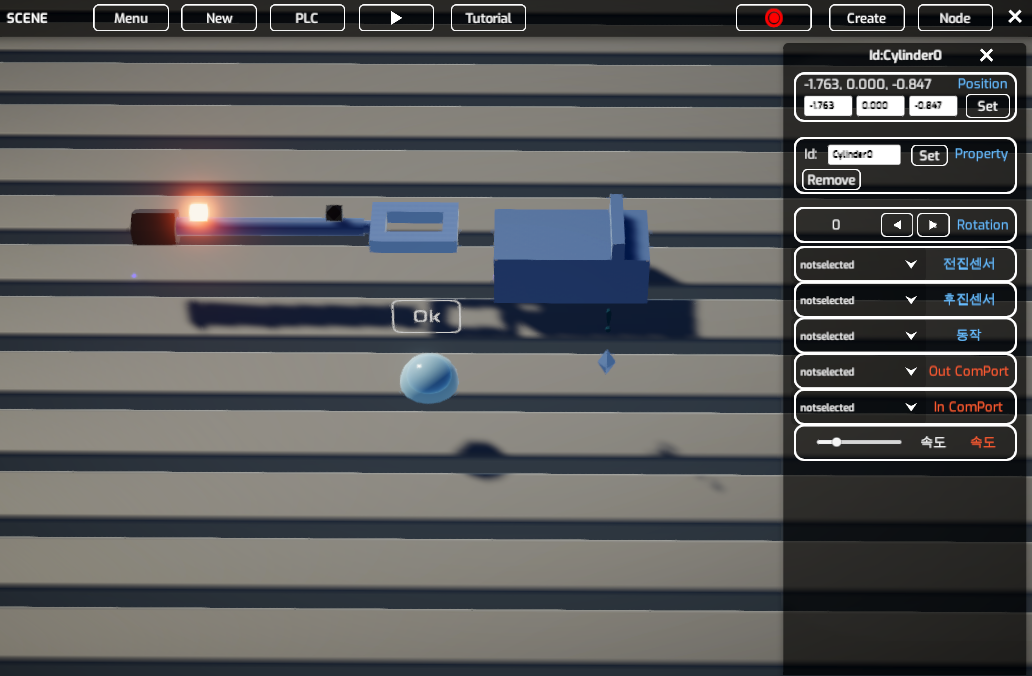
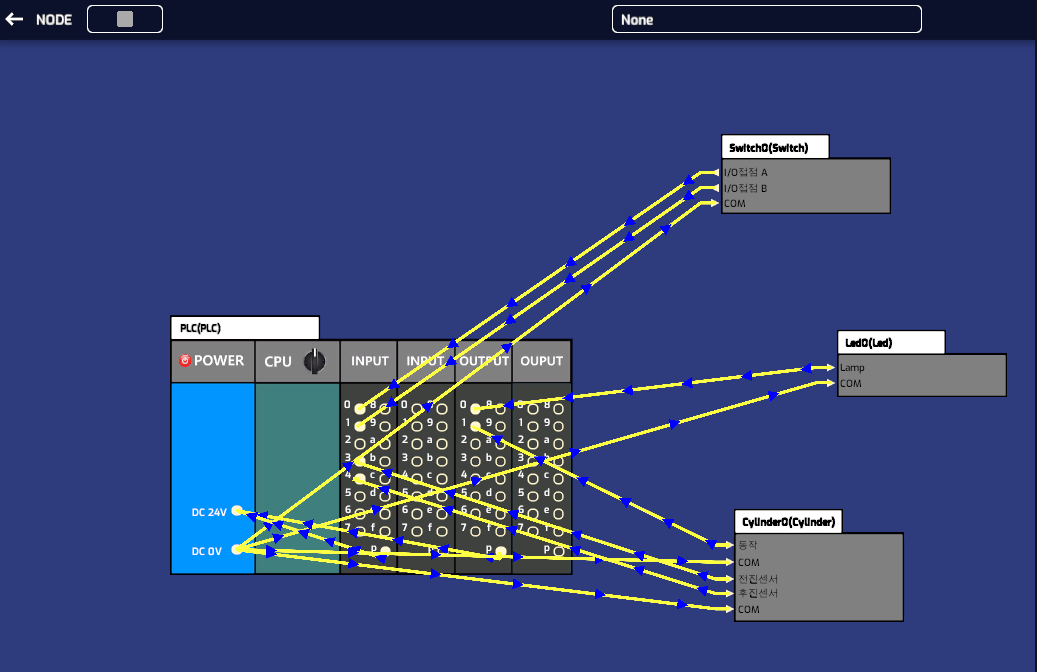
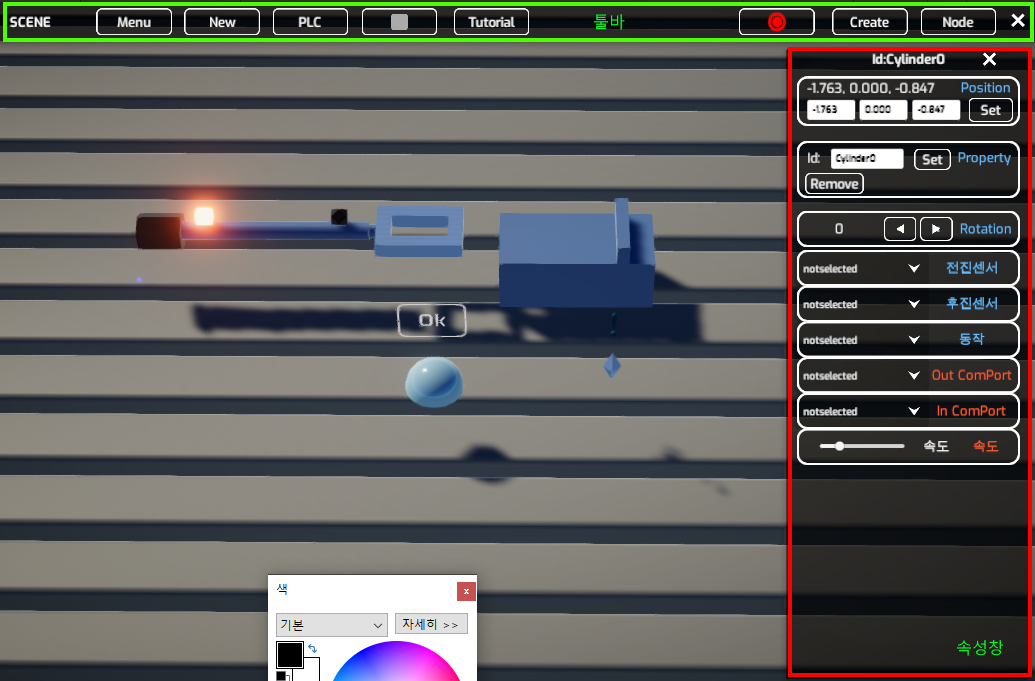
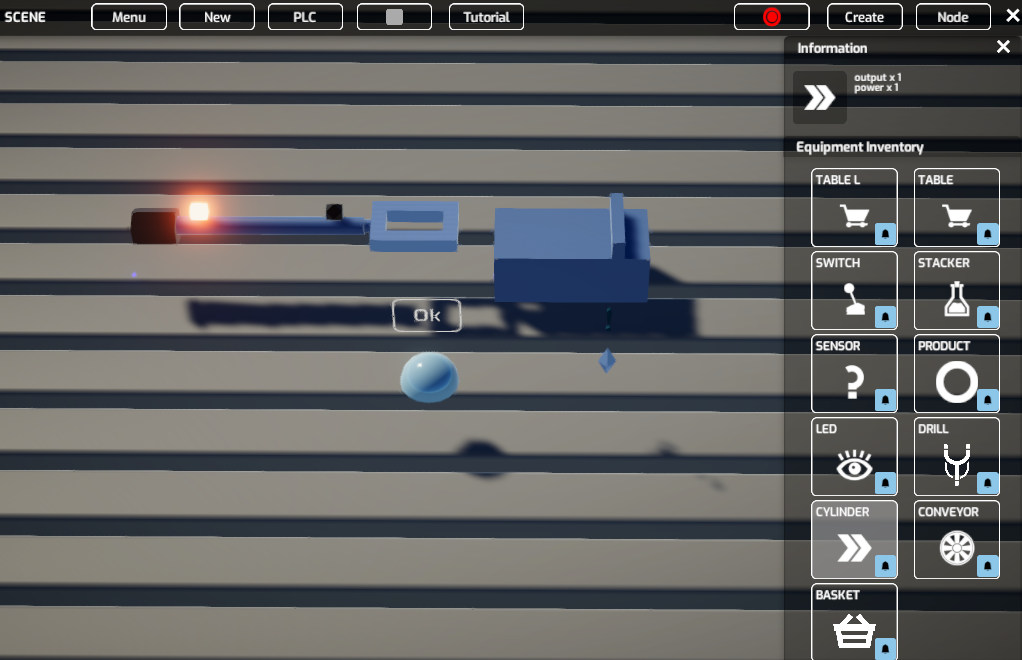
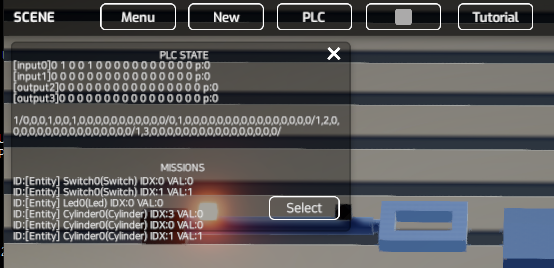
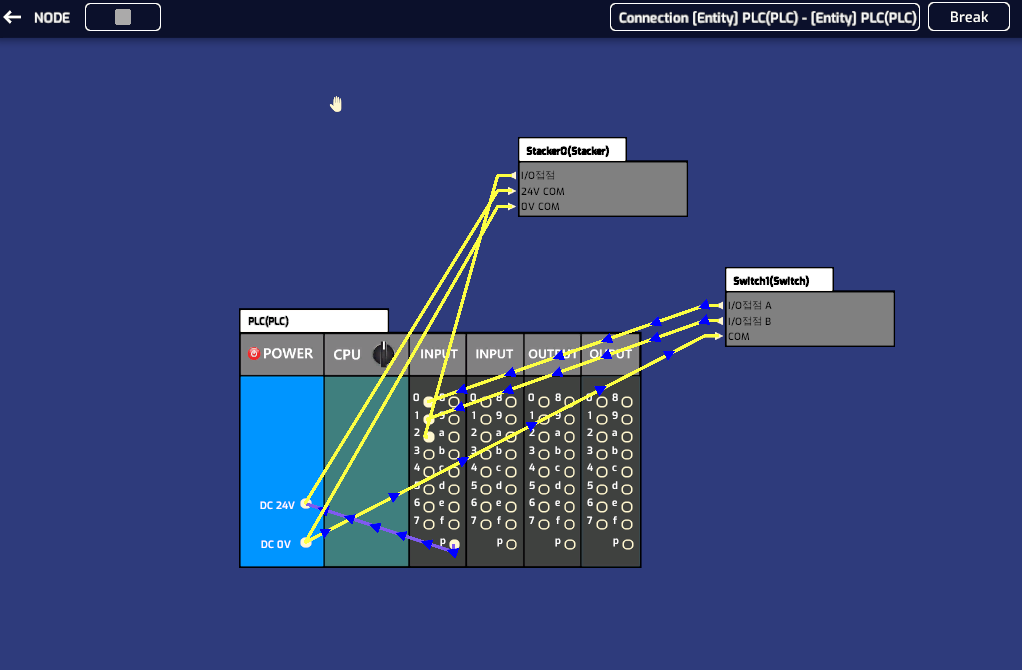
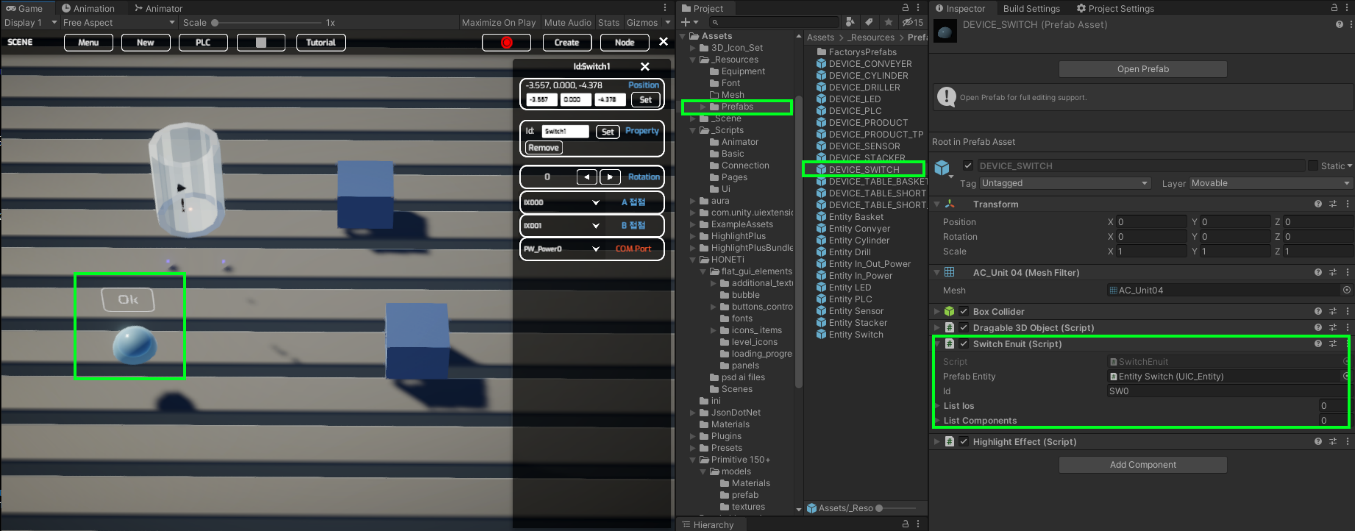
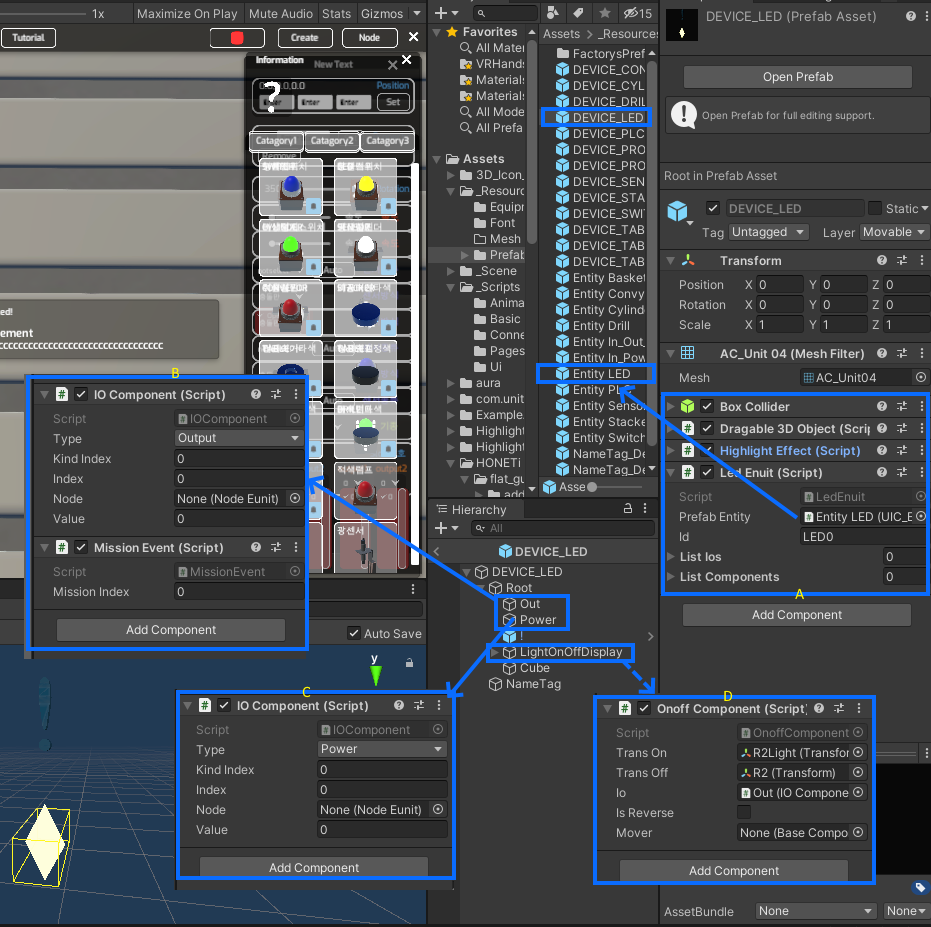
EUnit TOOL 구조 설명서

Eunit Tool의 구성은 크게 두 가지로 구성됩니다.

1. **Scene 뷰** 
   1. 씬뷰에서는 3d오브젝트 장치들을 배치하고 속성을 정의합니다. 실재 작업하는 것 것과 같은 화면 속에서 각각 장치들을 원하는 위치에 배치합니다.
   2. 
2. **Node 뷰**
   1. 노드뷰에서는 각각의 장치간의 연결관계를 중심으로 보여주는 화면입니다. 장치간의 연결, 전기의 흐름 등을 보여줍니다.
   2. 
3. **Scene 씬뷰의 구성** 
   * + 1. 
       2. 이씬에서는 장치들을 배치하고 장치의 유기적인 관계를 연결하고 작동시키는 설정을 주로 하는 씬 입니다. 인벤토리에서 장치들을 고르고 배치하며, 각 장치의 속성값을 입력하고 테스트 해보는 역할을 합니다
       3. 씬뷰는 상단 툴 바와 오른쪽 속성창의 구성으로 되어 있습니다. 상단 툴 바는 전반적인 기능들을.. 오른쪽 속성 창(inspect)은 각각의 장치들의 디테일한 속성 등을 설정합니다.
       4. 상단툴바는 **ToolBarPage** 클래스가 관리하며 오른쪽 속 성창은 **PageInspector** 클래스가 관리합니다.
       5. 툴 바의 Create버튼을 누르면 인벤토리 창이 뜹니다. 인벤토리 창은 **PageInventory** 클래스가 관리합니다. 인밴토리 창에서는 작업할 장치들의 목록을 보여주고 이를 월드 화면에 배치할 수 있게 해줍니다. 드래그해서 왼쪽 공간에 배치하며, 똑같이 노드뷰에서도 같은 엔티티가 생성이 됩니다.
       6. 
       7. **Menu버튼**은 초기 메뉴 화면으로 돌아갑니다. 튜토리얼, 연습 모드 등의 메뉴가 나옵니다.
       8. **PLC버튼**을 누르면 PLC의 상태를 보여주는 창이 나옵니다.
          1. 
       9. **Play/ Stop** 버튼을 클릭하면 Play Stop 모드로 반복 전환 됩니다. Play 상태 일 때 전류가 흐른다고 가정하여 장치들이 작동을 하게 됩니다.
          1. 
       10. **튜토리얼버튼** 을 누르면 현재 진행중인 튜토리얼을 설명을 다시 볼 수가 있습니다.
       11. **녹화버튼**을 누르면 현재 화면을 동영상으로 녹화합니다. 한번 더 누르면 녹화가 종료됩니다.
           1. 
       12. **Node**버튼을 누르면 Node뷰 화면으로 바뀝니다.
4. **Node 뷰 구성**
   1. 
   2. 씬뷰와 마찬가지로 상단 툴 바가 있습니다. **ToolBarNodePage** 클래스에서 관리합니다.
   3. 각각의 엔티티 항목들을 드래그 해서 배치 할 수 있으며 각각의 접점에서 서로 연결하여 PLC 와 연결을 할 수 있습니다.
   4. PLC와의 연결을 구성하는 것을 하는 씬이며, Power연결을 하여 전류를 흐르게 하는 것과 정점간의 연결을 통해서 장치와 PLC간의 전기의 신호를 보내는 구성을 만드는 것이 주로 이씬에서 하는 일입니다.
   5. 엔티티나 연결 선을 선택을 하면 Break 버튼이 보여지면 이 버튼을 눌러서 각 항목을 삭제 할 수 있습니다.
   6. 여기서도 똑같이 Play/Stop 버튼으로 전류를 흐르게 또는 멈추게 할수 있습니다.
5. **장치 오브젝트** 
   1. 
   2. 장치 오브젝트는 씬에 배치되서 유기적으로 작동되는 하나의 장치들을 말합니다. 각각의 장치들은 서로 연결되서 데이터 값에 의해서 작동되거나 PLC로 값을 전달합니다.
   3. **하나의 장치 오브젝트**는 다음으로 구성됩니다.
      1. **EquipmentEnuit** 장치의 인식 자
         1. EquipmentEnuit로부터 상속된 LedEnuit, SensorEnuit 등등.
         2. 모든스크립들을 EquipmentEnuit에 들어있고 상속을 받아서 Type을 정하는 식별 자 역할을 한다.
      2. **Dragable3DObject** 마우스로 잡아서 배치할 수 있는
         1. 마우스로부터 드래그 기능을 제공하기 위해서 OnMouseUp, OnMouseDrag등의 이벤트를 처리.
         2. 마우스로 드래그 중 Avoid 기능을 제공한다. 다른 개체와 충돌 시 서로 침범하지 않는 기능을 제공.
      3. **HighlightEffect** 아웃라인선택 표시
         1. 아웃라인 처리하는 쉐이더효과 plugin 구입하여 사용
         2. 오브젝트 포커스상태에서 아우라인효과를 보여주는 역할.
      4. **UIC\_Entity** 노드 씬에서 UML 형식으로 보여주는 UI개체
         1. 이 개체는 하위 개체로 있는 것이 아닌 Node Scene에서 독립적인 존재, 하지만 접점들이 연결되어 있다.
         2. Entity는 여러 개의 UIC Node로 구성되어 있고 **UIC Node는 NodeEunit**로 래핑되어 있다.
         3. UIC\_NODE는 UIC의 이름으로 시작되는 패키지로 UML스타일의 UI를 제공하는 플러그 인이다.
      5. **IOComponent Node** Node로부터 연결되어 값을 전달
         1. PLC로부터 값이 연결된 이곳으로 전달된다. 각각의 Node는 NodeEunit로 대응된다.
         2. 통상 IOComponent 는 **input**(PLC로 입력신호를 보냄) **output** (PLC로부터 신호를 받음 ) **power** (전력의 흐름을 제어하기 위한 접점)
      6. **작동제어 콤포넌트들** 하위 3D오브젝트들 작동시키는 콤포넌트
         1. PLC에서 오는 신호에 의해서 불이 켜지거나 컨베이어 벨트가 움직이거나 하는 등의 콤포넌트.
         2. 3d개채를 껐다 켰다 하는 **OnoffComponent**, 3d개체를 움직이는 MoveComponent, 3d 개체를 회전하는 **RotateComponent**, 애니메이션을 작동시키는 **AnimatorComponent**, 충돌을 검사하는 CollisionComponent등이 있다.
      7. **CheckAvoid** 충돌을 체크하여 겹치지 않게 방지
         1. Collision Check 을 해서 다른 오브젝트와 충돌을 체크한다. **Dragable3DObject** 등에서 물체를 움직일 때 사용
         2. 하위 개체 중에 충돌박스가 설치된 오브젝트에 추가한다.
      8. **LinkObject (NameTag)**로 개체의 이름과 수동 동작 버튼 표시
         1. Link오브젝트를 이용하여 canvas UI 개체를 서로 연결하여 Text,Button 등을 보여준다.
         2. 하위 오브젝트로 포함하는 것이 아닌, 지속적으로 위치를 따라다니도록 되어있어서 회전 등에 영향을 받지 않도록 되어 있다.
      9. **Snap Point** 로 근처에 있는 오브젝트들 붙게 만들기
         1. SnapPoint 의 콜라이더끼리 충돌검사를 해서 서로 붙게 만드는 Snap기능을 제공
         2. SnapPoint 스크립트와 Sphere 콜라이다가 추가 해야 함.
6. **Prefab화된 장치와 생성 과정** 
   1. 
   2. 모든 장치는 프리팹화 되어 저장된다 프리팹이 인스턴스화 되면서 PLC와 유기적으로 작동되는 장치들이 생성된다.
      1. 하위로 존재하는 콤포넌트들이 이 장치의 움직이고 작동하는 개체로서의 특징적인 역할을 한다.**(D)**
      2. 또한 이 작동하기 위한 신호를 받기 위해서 **IOComponent** Node이곳을 참조하는데 이곳은 PLC에서 주는 데이터를 받는 창구 역할을 한다. 이 데이터를 받은 구조는 Node로 연결된 PLC로부터 구조이다. **(B,C)**
         1. 위의 그림을 보면 **두 개의 IOCompoent** 가 있다 **하나는 Output** 그리고 또 하나는 **Power** 이다. Output 을로부터 신호를 받는데 이것은 PLC로부터 받는다. Power는 전류 연결에서 사용하는 IOComponent이다.

|  |
| --- |
| 단순히 데이터를 받는 역할만 한다. |
| public class IOComponent : MonoBehaviour  {  public IoType Type;  public int KindIndex;  public int Index;  public NodeEunit Node;  //public byte Value { get; set; }  public byte Value;  public void Receive( byte data )  {  Value = data;  }  public void Send( byte data )  {  }  } |

* + - 1. 장치를 작동시키기 위한 콤포넌트 **OnOff Component** 가 존재한다. 이것은 IOComponet로부터 값이 1 이되면 불을 켜주고 0이되면 불을 꺼주는 역할을 하는 콤포넌트 이다.

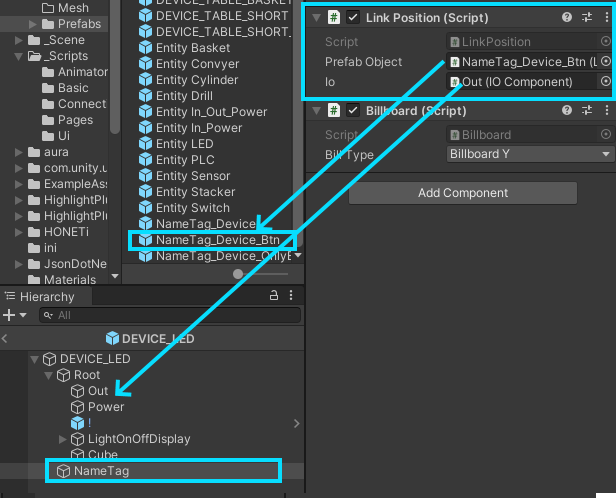
|  |  |
| --- | --- |
|  | 1. 위 개체에 서로 껐다 켰다 하는 오브젝트들을 가지고 있다. (TransOn, TransOff)  2. 그리고 이것은 IOComponent와 연결된 값에 의해서 결정된다. (Io)  3. 유니티의 gameObject 의 setActive 함수를 통한 활성화/비활성화 효과이다. On일대 켜지는 하위 개체들에 Light 가 있다면 불이 켜진 효과가 있을 것이다. |

* + - * 1. 해당 소스를 잠깐 참조하자.

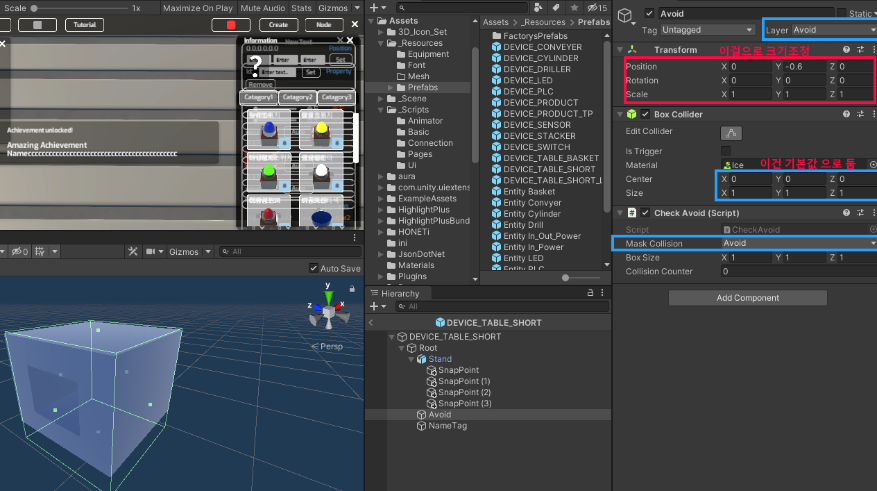
|  |
| --- |
| 단순히 값에 의해서 꺼주고 켜주는 역할  Reverse 를 체크하여 반대로 할 수도 있게 한다. |
| public class OnoffComponent : BaseComponent  {  public Transform transOn;  public Transform transOff;  public IOComponent io;  public bool isReverse;  public BaseComponent mover;    // Start is called before the first frame update  void Start()  {  }  void Update()  {  if (io != null)  {  //Debug.Log("===>" + io.Value + "==>" + isReverse );  transOn.gameObject.SetActive(io.Value == ((isReverse == false) ? 1 : 0) );  transOff.gameObject.SetActive(io.Value != ((isReverse == false) ? 1 : 0) );  }  if(mover != null )  {  transOn.gameObject.SetActive(mover.Handle == ((isReverse == false) ? 1.0f : 0.0f) );  transOff.gameObject.SetActive(mover.Handle != ((isReverse == false) ? 1.0f : 0.0f) );  }  }  } |
|  |

* + - 1. 그리고 이 프리팹화된 오브젝트는 **인벤토리(Inventory)**메뉴에서 드래그해서 생성한다. 다음과 같은 일들이 발생한다.
         1. 기본적으로 생성(인스턴스화)하는 오브젝트는 **장치프리팹** 자신과, **Entity**이다. 이 Entity는 프리팹에서 연결된 **Entity프리팹**을 으로 인스턴스화 된다. (Entity의 생성은 NodeManager에서 한다. )
         2. 그리고 Entity의 각각의 정 점점들과 하위개체에 있는 정 점들간의 연결을 한다. 서로 연결점을 가지게 하는 작업이 이루어 진다.
         3. 그리고 자신의 이름을 생성한다. **자동화된 이름 규칙에 의해서 생성**한다.

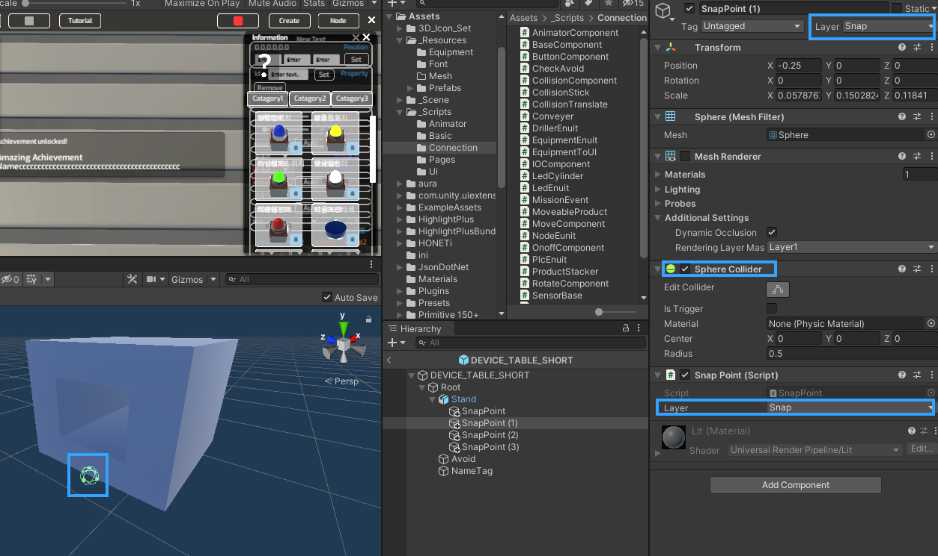
|  |
| --- |
| **장치와 Node를 연결 짓는 아주 중요한 코드 중 하나이다.** |
| virtual protected void Create()  {  **// 기본적인 이름 규칙에 의해서 이름을 자동 생성한다.**  if (idSequence.ContainsKey(Type) == false)  idSequence.Add(Type, 0);    Id = Type.ToString() + idSequence[Type];  **// PLC 는 그냥 고정이름.**  if (Type == EunitType.PLC)  Id = Type.ToString();  ++idSequence[Type];  **// 밑에있는 작동 콤포넌트들을 가져온다. IOComponent**  ListComponents = new List<BaseComponent>(GetComponentsInChildren<BaseComponent>(true));  for (int i = 0; i < ListComponents.Count; ++i)  ListComponents[i].Equipment = this;  Dragger = GetComponent<Dragable3DObject>();  if (PrefabEntity == null) return;  **// Node와 연결을 한다. Scene IoComponent과 NodeEunit들을 연결한다.**  ListIos = new List<IoElement>();  DicPower = new Dictionary<int, IoElement>();  **// Entity 를 먼저 생성한후**  Entity = PageNodeManager.Instance.CreateEntitny(PrefabEntity, Id, Type, Vector3.zero );  **// Entity 밑에있는 노드들을 가져온다.**  var nodes = Entity.gameObject.GetComponentsInChildren<NodeEunit>(true);  var ios = GetComponentsInChildren<IOComponent>(true);  **// 장치와 Entity의 노드들을 서로 연결한다.**  for( int i=0; i<ios.Length;++i)  {  var io = ios[i];  if (io.Type == IoType.Dummy) continue;  var node = FindNode(nodes, io.Type, io.KindIndex, io.Index);  var ioElement = new IoElement()  {  Kind = io.Type,  Index = io.Index,  KindIndex = io.KindIndex,  IoUnit = io,  NodeUnit = node  };  io.Node = node;  if (node == null)  Debug.Log("CannotFound node:" + gameObject.name + " type:" + io.Type + " Index:" + io.KindIndex + "/" + io.Index);  node.IoEnuit = io;  node.Equipment = this;  ListIos.Add(ioElement);  **// Power같은경우 Link를 만들어 딕쳐너리에 보관한다. 나증에 바로바로 찾으려고 함이다.**  if (io.Type == IoType.Power)  DicPower.Add(io.KindIndex, ioElement);  }  **//이렇게 모든 점점들의 리스트를 가지고 있고 소팅.**  ListIos.Sort(  delegate (IoElement left, IoElement right)  {  var valueLeft = (int)left.Kind \* 1000 + left.KindIndex \* 100 + left.Index;  var valueRight = (int)right.Kind \* 1000 + right.KindIndex \* 100 + right.Index;  return valueLeft.CompareTo(valueRight);  }  );   * + - * 1. } |
|  |

* + - 1. 하위개체등에서 **LinkPosition** 이 있다면 여기서 UI개체를 생성하고 연결한다. 그리고 지속적으로 이 위치와 연결되어서 이름 등을 표시한다. 버튼도 하나 제공되는데 이 버튼으로 해당 IOComponent의 값을 강제로 설정한다. PLC에서 적용된 값 말고 수동으로 설정하는 역할을 한다. 포커스가 되지 않는 상황에서는 어떤 값도 제어하지 않는다.
         1. 

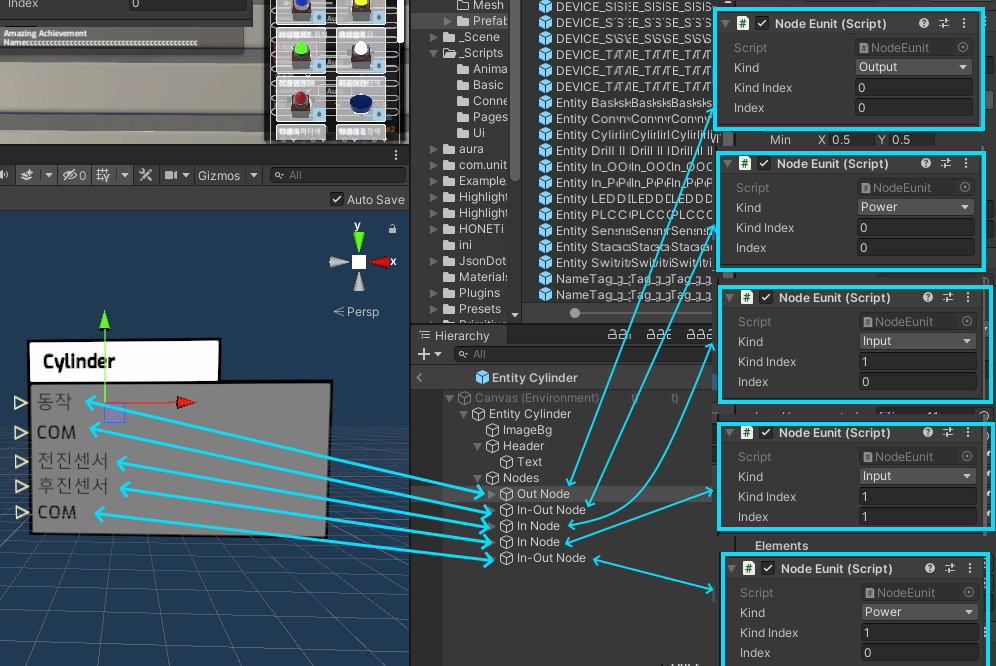
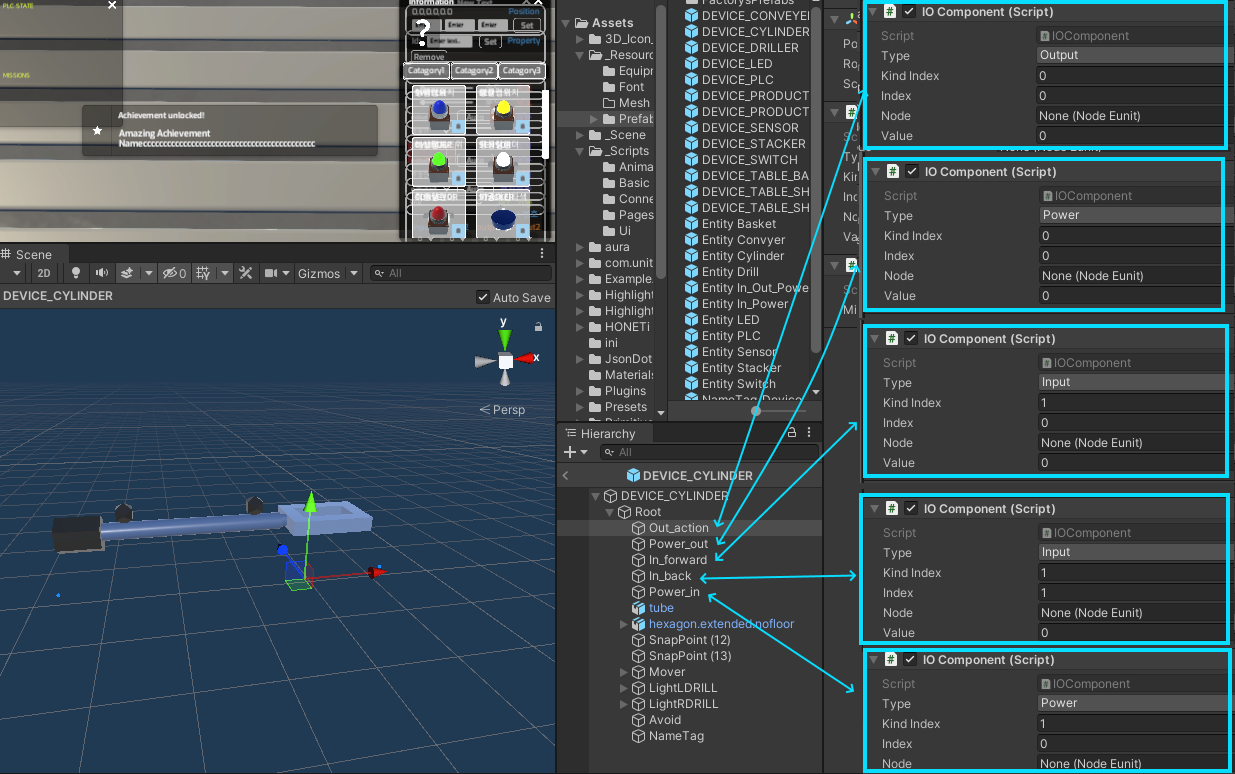
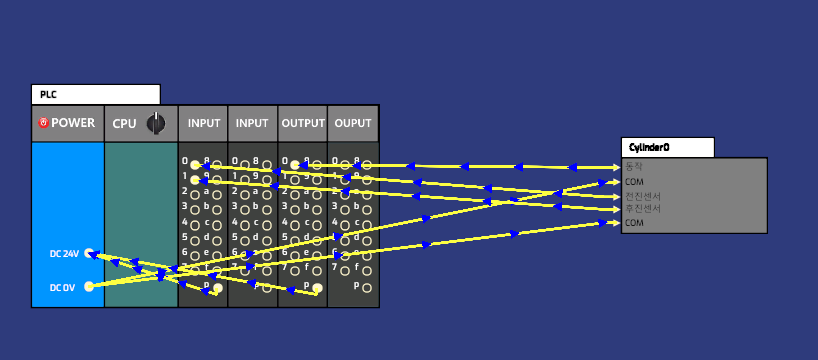
|  |
| --- |
|  |
| public class LinkPosition : MonoBehaviour  {  public LinkObject prefabObject;  public IOComponent io;    EquipmentEnuit equipment { get; set; }  private void Start()  {  **//LinkObject를 생성 (UI 개체) 한다.**  var newTag = Instantiate(prefabObject);  **//장치를 받아온다.**  equipment = GetComponentInParent<EquipmentEnuit>();  **//UI개체의 이름을 장치의 이름으로 설정하고, 버튼과 연결된 작용 IOComponent에 값을 설정을한다.**  newTag.transform.parent = transform.parent;  newTag.objectFrom = gameObject;  newTag.Text = equipment.Id;  newTag.CallbackState = NameTagBtnPress;  }  **//작동 버튼에 대한 스크립트이다. 포커스 중일때만 작동하고 버튼이 눌리고 있을 때는 계속 1 아닐때는 0의 값으로 IoCompont를 설정하고 있다. 이 값에의해서 장치들이 작동을 하고 있으므로 강제설정과 같은 역활을 한다.**  virtual protected void NameTagBtnPress(bool isFocus, bool isPress)  {  if (isFocus == false) return;  io.Value = (isPress == true) ? (byte)1 : (byte)0;  }  } |
| public class LinkObject : MonoBehaviour  {  public GameObject objectFrom;  public Vector3 offset;  public Text uitext;  public UiEunitButton btnUi;  public bool IsRotation;    public string Text  { get { return uitext.text; }  set { uitext.text = value; }  }  public UnityAction<bool, bool> CallbackState { get; set; }  = delegate (bool ispointed, bool isdowned) { };    TimeCall.TimeCallElement timer;  void Start()  {  **// 항상 위치를 연결 장치의 위치로 설정해주는 콜백함수 제작**  **// TimeCall 로 콜백함수를 따로 호출해준다. Update 가 많아지면 효율이 않좋기 때문**  timer = TimeCall.Instance.AddAlwaysCall  ( delegate ()  {  if (objectFrom == null) return;  transform.position = objectFrom.transform.position + offset;    if (IsRotation == true)  transform.rotation = objectFrom.transform.rotation;  }  );  **// 버튼을 누르면 콜백 함수를 호출한다. 이 콜백함수는 LinkPosition에서 설정한다.**  **// IOComponent의 값을 설정하는 역활을 한다.**  btnUi.callbackState = delegate (bool isPointed, bool isDowned)  {  CallbackState(isPointed, isDowned);  };  }  private void OnDestroy()  { timer.SetRemove(); }} |

* + - 1. 다른 오브젝트끼리 충돌을 피하기 위하여 **Check Avoid** 기능을 사용하는데 장치 오브젝트 하위에 이렇게 박스 콜라이더 하나만 추가해서 **Check Avoid**를 추가하면 된다. 하지만 형식을 좀 요구하는데 Layer는 Avoid 박스 콜라이더는 기본값만 사용, 충돌 체의 크기나 위치 조정은 Trasnform으로 한다. 그리고 충돌검사를 하는 소스에서 이 Check Avoid 을 참조하고 있어야 한다.
         1. 
         2. 이 충돌 체를 검사하여 위치 이동을 막는 스크립트는 다음과 같다.

|  |
| --- |
| Dragable3DObject.cs |
| **// 충돌검사를 스크립트에서 바로 하는 방법을 사용한다.**  **// 이렇게 하면 스크립트에서 간단하게 충돌방지를 처리할 수 있다.**  public bool IsAvoidCollision  {  get  { if (avoidBox == null) return false;  else return (avoidBox.CheckAvoidCollision() == true); }  }  public bool IsAvoidCollision2//bskim  {  get  { if (avoidBox2 == null) return false;  else return (avoidBox2.CheckAvoidCollision() == true);  }  }  void OnMouseDrag()  {  if (enabled == false) return;  if (IsSelected == false) return;    Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);  if (plane.Raycast(ray, out distance))  {  LastGroundPosition = ray.GetPoint(distance);    **// 현재위치를 백업하고 움직인다.**  var position1 = transform.position;  transform.position = LastGroundPosition + offset;    **//움직인 위치에서 Avoid충돌체 끼리 충돌이 있는지 검사**  **//충돌되면 움직이기 이전위치로 돌린다. ( 마지막 충돌이 발생하지 않은 위치)**  if( IsAvoidCollision == true || IsAvoidCollision2 == true) //bskim  transform.position = position1;    **// 움직일수 있는 월드 위치 제한**  if( transform.position.x < -42.0f || transform.position.x > 40.0f ||  transform.position.z < -55.0f || transform.position.z > 30.0f )  transform.position = position1;    if (eunit != null)  eunit.UpdateComponents();  }  } |
| CheckAvoid.cs 클래스에서 충돌 검사하는 소스는 다음과 같다.  public bool CheckAvoidCollision()  {  collisionCounter = 0;  **// Physics.OverlapBox 를 이용해서 바로 충돌검사 스크립트에서 바로한다.**  Collider[] hitColliders =  Physics.OverlapBox(transform.position, transform.lossyScale \* 0.5f, transform.rotation, maskCollision);  **// 나자신과의 충돌은 배제한다.**  for (int i = 0; i < hitColliders.Length; ++i)  {  if (hitColliders[i] == collider) continue;  ++collisionCounter;  }  return collisionCounter > 0;  } |

* + - 1. Snap Point 기능은 장치오브젝트 하위에 SnapPoint를 추가만 하면 근접거리에서 서로 붙는 Snap기능을 사용할 수 있다. Snap Point를 그냥 복사하기로 밑에 붙이기만 해도 된다. 스냅포인트는 장치개체 하위에 다음과 같이 존재한다.
         1. 

|  |
| --- |
| SnapPoint.cs |
| public void Update()  {  **// 현재 선택중인 오브젝트들만 대상으로 한다.**  if (FirstPlaceObject.Selected == null)  {  if (Equipment.IsSelected == false) { other = null; return; }  }  else  if (FirstPlaceObject.Selected != Equipment.GetComponent<FirstPlaceObject>())  {  other = null; return;  }    **// 원형 오브젝트 충돌검사를 해서 충돌이 검출되면 대상 오브젝트로 설정한다..**  var colliders = Physics.OverlapSphere(transform.position, 0.1f, layer);    if(colliders.Length>0)  {  for (int i = 0; i < colliders.Length; ++i)  {  var snapPoint = colliders[i].GetComponent<SnapPoint>();  if (snapPoint == null) continue;  if (snapPoint == this ) continue;  var iscollision = layer.Contains(snapPoint.gameObject.layer);  if (iscollision == false) continue;    if (other == snapPoint) continue;  other = snapPoint;  }  }  }  private void LateUpdate()  {  if (other == null) return;  var distance = (transform.position - other.transform.position).magnitude;  **// 원래위치(현재 SnapPoint와 위치와 대상 오브젝트의거리) 에서 벌어지면 스냅기능을 취소한다.**  if (distance > 0.5f)  other = null;  Else  **// 범위내 있다면 현재의 위치에서 붙는 기능을 활성화 한다. 장치오브젝트의 위치를 설정해야 하므로 상대적인 값을 계산하여 설정한다.**  Equipment.transform.position = other.Equipment.transform.position +  (Equipment.transform.position - transform.position)  - (other.Equipment.transform.position - other.transform.position);    } |

1. **Entity 와 장치간의 접점 연결**
   1. 장치오브젝트 하위 개체들에 IOComnponent 들이 있는 건 이미 위에서 설명을 했다. 이것들과 똑같은 접점들의 연결이 존 제하는데 Entity에 포함되는 접점들( NodeEunit, UIC\_Node ) 연결이 된다. 이것은 정확히 개수와 형식이 일치 해야 한다.
   2. 이는 장치오브젝트 Create 과정에서도 과정이 있다.
   3. Entity 의 접점 구성
      1. 
      2. 위 그림처럼 각각의 Entity의 접점 노드들은 각각 NodeEunit 과 연결되며 **NodeEunit은 Kind(점점 종류)와 KindIndex, Index**로 구성된다. 이것들 3가지가 서로 일치 하는 것끼리 연결이 된다.
         1. **Kind** : 정점의 종류 Input(PLC에게 신호를 주는 것) Outpu(PLC신호를 받는 것) Power(전기연결)
         2. **KindIndex** : 장치 내 내부장치 구분에 사용, 장치 내 작동 부와 센서 부 처럼 두 가지 하드웨어가 연결되어 있는 경우가 있고, Power도 두 개가 존재한다. 하나의 Power당 하나의 장치를 작동시키기위한 전력으로 작동되므로 이 **Power별 구별을 KindIndex로 한다**. 위의 실린더를 보면 작동 부 는 KindeIndex 가 0으로, 센서부는 KindIndex 가 1로 설정되어 있다.
         3. **Index** : 접점의 식별 자로 사용된다. 같은 점점 형식을 가진 개체가 두 개 이상 있으면 안되기 때문에 고유 식별 자로 사용한다.
   4. 장치의 점점 구성
      * 1. 
        2. NodeEunit와 1:1 대응되는 IOComponent가 똑같이 구성됨을 알 수가 있다.
        3. 이들은 서로 1:1 연결되어 있고 값을 공유한다.
   5. PLC 장치로부터 통신
      1. Entity가 따로 존재하는 이유는 PLC 과의 연결을 정의 하기 위함이다.
      2. 장치들에서 PLC 연결을 하기에는 연결이 너무 복잡해질 수가 있고 도식화된 연결을 보여주기 위해서 따로 연결 방법을 제공하는데 그것에 사용되는 게 Entity 이며, NodeManager 에서 이를 관리한다.
      3. PLC로 신호를 보내면 PLC 는 외부 프로그램과 연결하여 데이터를 전달하며, 또한 그 프로그램으로 데이터를 받아서 내부 장치들에게 값을 전달한다. 이 중간 전달 과정에 Entity가 역할을 한다.
      4. PLC Entity 와 Cylinder Entity 간의 연결 구성
         1. 
         2. 각각 Input, Output, Power 가 서로 연결되어 있다.
         3. **Entity** 와 접점은 **UIC\_Node**(연결 부)가 구성됨, UIC\_Node 간의 연결(라인연결)은 **UIC\_Connection** 으로 구성된다. 다 UIC\_로 정의 하는 이것들은 UI Node Connect라는 플러그 인에서 제공하는 소스이다. 이 소스들에 대한 설명은 생략합니다. 단 **UIC\_Node 에 NodeEunit** 를 추가해서 서로 공유하도록 구성합니다.
         4. PLC는 **PLCEunit** 로 구연되어 있으며 역시 EquipmentEnuit로 부터 상속받아서 구성되며 EquipmentEnuit 으로 상속을 받아서 가장 많은 작동 코드가 구연되어 있다.
2. **PLC 구동 PLCEunit**
   1. PLC역할을 하는 PLCEunit 에서 하는 작업들은 다음과 같다.
      1. **각 장치들간에 전력을 공급한다.**
         1. 기본 0v, 24v 의 전력 공급 접점이 있고, 이와 연결할 수 있는 Power가 4개 있다.
         2. 그리고 파워가 제대로 연결되어 있는지 검사하며 제대로 연결되어 있다면 작동할 수 있게 한다.
      2. **장치들로부터 받는 신호를 외부의 프로그램에게 전달한다.**
         1. 주로 입력장치들이 해당한다. 센서, 스위치 등.
      3. **외부프로그램으로 부터 받은 신호를 장치들에게 전송한다.**
         1. 주로 출력장치들에 해당한다. 램프, 실린더 등.
   2. **전력공급** 
      1. 모든 접점들을 검사, 전력 공급이 가능한지 판단
      2. 전력공급이 가능할 때 라인 애니메이션.
      3. 전력 공급이 가능하면 데이터가을 전송한다.
      4. 다음 소스를 참조하자

|  |
| --- |
| PLCEunit.CS 내용 |
| public void UpdateSlots()  {  **// PLC 내의 접점들을 모두가 검사해본다.**  for (int i = 0; i < ListIos.Count; ++i)  {  var slotOne = ListIos[i];  **// Power 접점들을 검사해볼필요가 없으므로 패스**  if (slotOne.NodeUnit.Kind == IoType.Power ||  slotOne.NodeUnit.Kind == IoType.Power0 ||  slotOne.NodeUnit.Kind == IoType.Power24)  continue;    **// 이 접점과 연결되어 있는 Power접점부의 파워 연결을 파악**  var powerType = DicPower[slotOne.KindIndex].NodeUnit.EndConnectType();    **// play 중이 아니라면 전원 공급이 안되는걸로..**  if (PageSceneManager.Instance.IsPlay == false)  powerType = IoType.None;  **// 파워 점접부로 연결되는 과정에서 연관된 Connection과 노드(NodeEunit)들과 Entity들을 얻어온다.**  var connectionPower = DicPower[slotOne.KindIndex].NodeUnit.connectionFlow;  var nodesPower = DicPower[slotOne.KindIndex].NodeUnit.nodeFlow;  var equipmentsPower = DicPower[slotOne.KindIndex].NodeUnit.entitiyFlow;    **// 접점부와 연결된 장치의 파워 연결을 파악**  **// 장치와 연결되는 과정에서 연관된 Connection과 노드(NodeEunit)들과 Entity들을 얻어온다.**  var slotLastType = slotOne.NodeUnit.EndConnectType();    var connections = slotOne.NodeUnit.connectionFlow;  var nodes = slotOne.NodeUnit.nodeFlow;  var equipments = slotOne.NodeUnit.entitiyFlow;    var speed = 0.0f;    **// PLC파워 접점부과 장치(Entity) 의 파워 접점을 파악한다. 서로 극성이 다른경우 전기가 통하는 것로 판단**  **// 전기가 흐르고 있다면 전기의 흐르는 방향도 파악**  if (powerType == IoType.Power0 && slotLastType == IoType.Power24)  speed = 0.1f;    if (powerType == IoType.Power24 && slotLastType == IoType.Power0)  speed = -0.1f;    **// 전기가 않르르고 있다가 +방향으로 흐르는 시점에**  if (slotOne.NodeUnit.IsPowerConnect == false && speed != 0.0f)  {  **// 스위치일경우 모든 전기흐름 표시를 해제하고 다시 설정.**  if (nodes.Count > 0 && nodes[0].Equipment.Type == EunitType.SensorCollision)  ClearConnection(nodes[0].UicNode.entity);    ++DicPower[slotOne.KindIndex].NodeUnit.PowerUseCount;    for (int k = 0; k < connections.Count; ++k)  SetConnectionLine(connections[k], nodes[k].UicNode, -speed);    for (int k = 0; k < connectionPower.Count; ++k)  SetConnectionLine(connectionPower[k], nodesPower[k].UicNode, speed);    for (int k = 0; k < equipments.Count; ++k)  equipments[k].IsPower = true;    slotOne.NodeUnit.IsPowerConnect = true;  }  Else  **// 전기가 않르르고 있다가 -방향으로 흐르는 시점에**  if (slotOne.NodeUnit.IsPowerConnect == true && speed == 0.0f)  {  **// 스위치일경우 모든 전기흐름 표시를 해제하고 다시 설정.**  if (nodes.Count > 0 && nodes[0].Equipment.Type == EunitType.SensorCollision)  ClearConnection(nodes[0].UicNode.entity);    --DicPower[slotOne.KindIndex].NodeUnit.PowerUseCount;    for (int k = 0; k < connections.Count; ++k)  SetConnectionLine(connections[k], nodes[k].UicNode, 0.0f);    **// 더이상 연결이 없을경우 전기 흐르지 않는걸로 표시**  if (DicPower[slotOne.KindIndex].NodeUnit.PowerUseCount < 1)  for (int k = 0; k < connectionPower.Count; ++k)  SetConnectionLine(connectionPower[k], nodesPower[k].UicNode, 0.0f);    for (int k = 0; k < equipments.Count; ++k)  equipments[k].IsPower = false;    slotOne.NodeUnit.IsPowerConnect = false;  }    **// input일 경우엔 현재 plc 에 값을 업데이트**  if (slotOne.Kind == IoType.Input)  {  **// 전기가 흐르고 있다.**  if (speed != 0.0f)  {  **// 전기가 흐른다면 PLC접점 값이 1로 설정, 일단 전기가 흐르니까**  slotOne.NodeUnit.IoEnuit.Value = 1;    **// 그후 접점과 연결된 장치로 부터 값을 받는다.**  for (int j = 0; j < nodes.Count; ++j)  if (nodes[j].Kind == IoType.Input)  slotOne.NodeUnit.IoEnuit.Value = nodes[j].IoEnuit.Value;  }  else  slotOne.NodeUnit.IoEnuit.Value = 0;  }    **// output일 경우엔 모든 연결 노드에 값을 준다.**  if (slotOne.Kind == IoType.Output)  {  **// 전기가 흐르고 있지 않타면 0을 기본값으로 주고, 흐른다면 외부 프로그램에서 받은 값(Output인경우)**  **// 을 각 장치에 값을 준다.**  var Value = (speed == 0.0f) ? (byte)0 : slotOne.NodeUnit.IoEnuit.Value;    for (int j = 0; j < nodes.Count; ++j)  if (nodes[j].Kind == IoType.Output)  nodes[j].IoEnuit.Value = Value;  }  }  } |

* + 1. 전기가 흐르는지를 판단하는 방법이 중요한데, **PLC의 접점의 Power**와 / **PLC 접점과 연결된 장치의 Power**를 어느 Power에 연결되었는지 추적해 봐야 한다. 이 추적된 값이 마지막이 PLC의 0V 나 24V 에 연결되어 있다면 마지막에 파워가 연결되어 있는 것이며, 각각 마지막 파워가 서로 달라야(극성이 달라야) 0V->24V 로 전기가 흐를 수 있는 것이다.
    2. 마지막 파워까지 추적하는 함수는 EndConnectType() 함수이다.

|  |
| --- |
|  |
| public IoType EndConnectType()  {  **// 현재 노드로 부터 연결된 노드들을 추적한다.**  int nodeCount = 0;  connectionFlow = new List<UIC\_Connection>();  entitiyFlow = new List<EquipmentEnuit>();  nodeFlow = new List<NodeEunit>();  PrevNodeFlow = null;  var connect = this;    do  {  **// 추적된 노드가 Power0, Power24 이면 이 값을 리턴... 올바른 파워 연결**  if (connect.Kind == IoType.Power0 || connect.Kind == IoType.Power24) return connect.Kind;  **//연결된 노드가 없다면 종료**  if (connect.GetConnectNode() == null) break;    **// 장치에서 연결된게 나의 노드 라면 장치의 Power로 추적한다.**  if (connect.PrevNodeFlow == connect.GetConnectNode())  connect = connect.GetPowerEntityNode();  else  {  **// 계속 추적해 나아간다. 다음 노드로 진행.. 이 진핸된 노드들을 저장해둔다.**  connect = connect.GetConnectNode();  connectionFlow.Add(connect.SelectedConnectFlow);  nodeFlow.Add(connect);  }    if (connect == null)  return IoType.None;  **// 장치들도 저장해 둔다.**  if (connect.Equipment != null && connect.Equipment != this.Equipment)  entitiyFlow.Add(connect.Equipment);    **// 무한루프를 방지.**  if(++nodeCount>10 ) return IoType.None;  }  while ( true );    return connect.Kind;  } |
| public NodeEunit GetPowerEntityNode()  {  **// 엔티티내 다른 Power개채를 찾는다. 본인의 타입이 power일경우는 본인으로**  if (Kind == IoType.Power ||  Kind == IoType.Power0 ||  Kind == IoType.Power24)  return this;  **// Node들을 검색한다.**  **// 이때 Power이고, KindIndex가 같은 파워를 찾는다.**  for ( int i=0;i< UicNode.entity.nodeList.Count; ++i)  {  var node = UicNode.entity.nodeList[i];  if (node == this) continue;    if (node.NodeEunit.Kind == IoType.Power ||  node.NodeEunit.Kind == IoType.Power0 ||  node.NodeEunit.Kind == IoType.Power24 )  **// KindIndex 가 같은 파워를 찾는과정 KindeIndex로 장치내 Power를 구분.**  if( node.NodeEunit.KindIndex == KindIndex )  { node.NodeEunit.PrevNodeFlow = null;  node.NodeEunit.SelectedConnectFlow = null;  return node.NodeEunit;  }  }  return null;  } |

* + 1. 전기가 흐르는 표시 중에 까다로운 점은 연결된 라인이 한 개가 아닐 수 있다는 점이다. 보통 PLC의 0v,24v 접점과 연결된 다른 접점들은 복수개가 생긴다. 그래서 전기의 흐름의 판단에 Counter 가 사용된다. 이 Counter가 0이 될 때 더 이상 전기가 흐르지 않는 것으로 판단한다.
    2. **센서**의 경우 내부에 파워 점점이 두 개가 존재하므로, 전기가 흐르지 않을 때 강제로 이 두 개의 파워 접점을 연결 상태를 흐르지 않음으로 설정해 줘야 한다. 연결되지 않는 노드는 아예 판단하지도 않기 때문이고, 스위치는 이 연결이 장치의 옵션에 따라서 수시로 바뀌기 때문이다.
    3. 라인의 애니메이션의 흐르는 효과는 SetConnectionLine() 함수의 호출로 이루어진다.

|  |  |
| --- | --- |
| public void SetConnectionLine(UIC\_Connection connect, UIC\_Node node, float speed)  {  if (speed == 0.0f)  { connect.line.animation.isActive = false;  return;  }  **//애니메이션 작동**  connect.line.animation.isActive = true;  **//애니메이션의 크기**  connect.line.animation.size = 10;  **//나오는 화살표갯수**  connect.line.animation.pointCount = 6;  **//색**  connect.line.animation.color = Color.blue;  **//흐르는 방향과 속도**  connect.line.animation.speed = connect.node0 != node ? -speed : speed;  } | |
|  |

* 1. **외부 프로그램에 데이터 전송**
     1. 외부 프로그램으로 데이터 전송은 COPY STRUCT 라는 윈도우 API 메시지 전달 방법이며 이 부분을 담당하는 소스는 **Moduler** 에서 한다
     2. PLC에서는 현재 접점 값을 스트링으로 변환하여 전송한다. 아래 소스는 PLC접점에서 값을 문자열로 받아오는 과정이다.

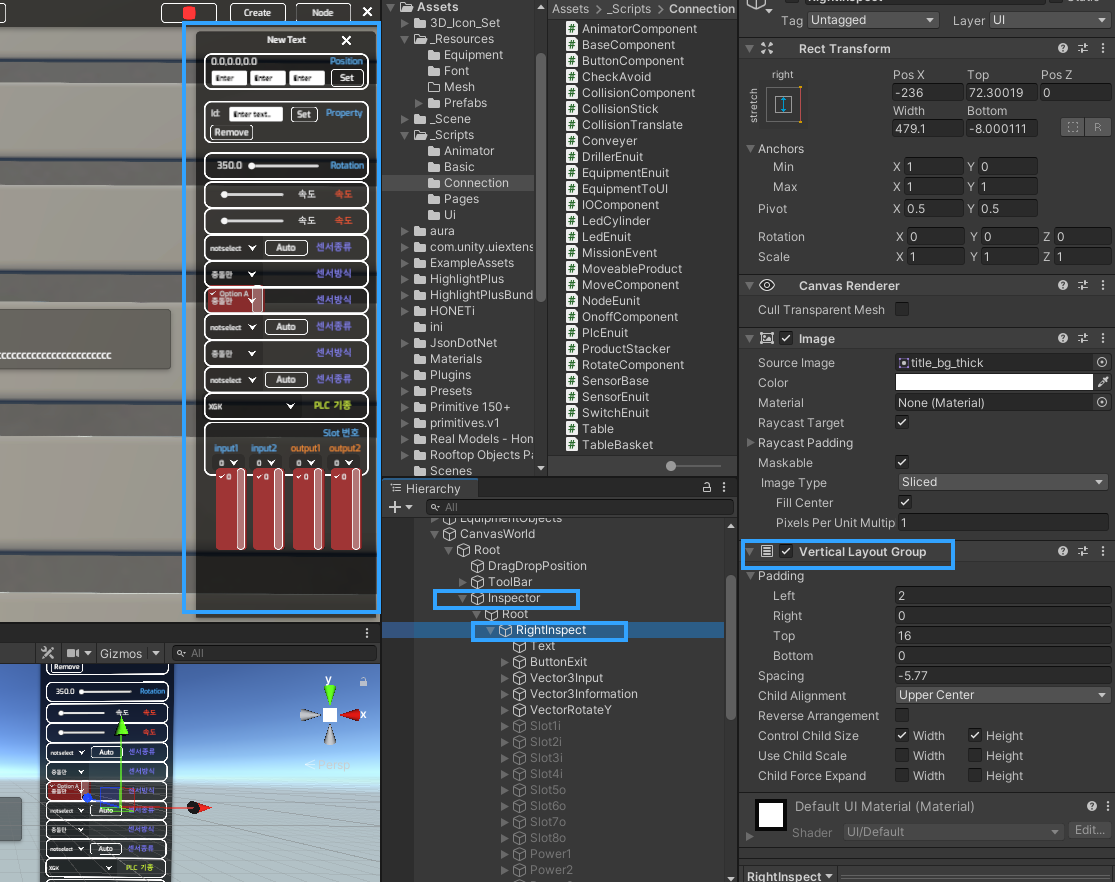
|  |
| --- |
| public void UpdateIoToString()  {  if (textInfo == null) return;    **// 스트링 빌더를 이용해서 무식한 문자열 사용 개선..**  **// dump 는 디버깅 메세지용**  StringBuilder dump = new StringBuilder();  **// share 가 실제 보내기 위한 문자열**  StringBuilder share = new StringBuilder();  /**/ PLC 형태 처음에**  share.Append( ((int)PlcType).ToString());  share.Append("/" );    **// 각 4개의 PLC 구릅을 검사한다.**  for (int j = 0; j < 4; ++j)  {  IoType typeIo = FindIo(IoType.Input, j, 0) != null ? IoType.Input : IoType.Output;  StringBuilder ioLine = new StringBuilder();    if (typeIo == IoType.Input) ioLine.Append($"[input{j}]");  else ioLine.Append( $"[output{j}]");  **// 인풋인지 아웃풋인지**  share.Append(((int)typeIo).ToString() +"," );    **// 각각의 구릅의 번호를 보냄**  if (typeIo == IoType.Input)  share.Append($"{IndexSlot[j]},");  else  share.Append($"{IndexSlot[j]},");  **// 16개의 접점들의 값을 추가**  for (int i = 0; i < 16; ++i)  {  var inputio = FindIo(typeIo, j, i);  ioLine.Append( $"{inputio.IoUnit.Value} ");  share.Append( $"{inputio.IoUnit.Value},");  }    var power = FindIo(IoType.Power, j, 0);  dump.Append(ioLine);  dump.Append($"p:{power.IoUnit.Value} \n");  **// 마지막에, 를 없애고 /을 추가**  share.Remove(share.Length - 1, 1);  share.Append("/");  }    textInfo = dump.ToString();  textShare = share.ToString();  //Debug.Log(textShare);  } |
| 다음과 같은 문자열 전송됨  0/  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1/  0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0/  1,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1/  1,3,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1/  이를 해석하면 아래와 같을 것이다.  Plc kind/  input or output, group index, value, value …../  input or output, group index, value, value …../  input or output, group index, value, value …../  input or output, group index, value, value …../ |

* + 1. 이문자열을 전송하는 소스는 다음과 같다

|  |
| --- |
| public void WriteToIPC( string value )  {  **// 너무 자주 전송하지 안토록..**  timeToSend += Time.deltaTime;  if (timeToSend > 0.1f)  {  timeToSend = 0.0f;  Moduler.Instance.Send(value);  }  } |
| public bool Send( string message )  {  return SendMsg("MetaAdapter", message);  } |
| **// Window Message 주고받는 방식으로 메세지 전송**  public bool SendMsg(string tagetProgram, string inMsg)  {  if (tagetProgram == null || tagetProgram.Trim() == "" || inMsg.Trim() == "")  {  return false;  }  try  {  **// 다른 외부 프로그램 을 찾음**  IntPtr handle = FindWindow(null, tagetProgram);  if (handle != null)  {  byte[] buff = Encoding.Default.GetBytes(inMsg);    **// COPY STRUCT를 방법으로 메세지 전달. (windows api 로 각 프로세스 간 전달)**  COPYDATASTRUCT cds = new COPYDATASTRUCT();  cds.dwData = IntPtr.Zero;  cds.cbData = buff.Length\*2+1;  cds.lpData = inMsg;    var pnt = SendMessage(handle, WM\_COPYDATA, 0, ref cds);  }  return true;  } |

* 1. **외부 프로그램으로 데이터 받기**
     1. 동일한 COPY STRUCT 방법으로 문자열 데이터를 받는다.

|  |
| --- |
| private IntPtr wndProc(IntPtr hWnd, uint msg, IntPtr wParam, IntPtr lParam)  {  if (Application.isFocused == false) return CallWindowProc(oldWndProcPtr, hWnd, msg, wParam, lParam);    try  {  switch (msg)  {  **// Window Message 로 받아서 데큐에 저장해 둔다.**  case WM\_COPYDATA:  {  que.Enqueue(Marshal.PtrToStructure<COPYDATASTRUCT>(lParam));  return DefWindowProc(hWnd, msg, wParam, lParam);  }  case WM\_Quit:  {  TryExecuteExit();  break;  }  }  return CallWindowProc(oldWndProcPtr, hWnd, msg, wParam, lParam);  }  catch (Exception)  {  return IntPtr.Zero;  }  } |
| public IEnumerator Dequeue()  {  while(isRunning)  {  if (que.Count > 0)  {  COPYDATASTRUCT cds = que.Dequeue();    if (cds.lpData != null)  TextMessage = cds.lpData;  **// 하나씩 꺼내서 콜백함수로 호출**  CallbackReceiveMsg(cds.lpData);    yield return null;  }  yield return new WaitForEndOfFrame();  }  yield break;  } |
| **//PLCEunit 에서 콜백함수로 자신의 GetFromIPC함수를 호출하도록 설정.**  Moduler.Instance.CallbackReceiveMsg = GetFromIPC; |
| public void GetFromIPC(string str)  {  textReadIpc = str;    if (IsTestMode == false)  { SetIoValue(str);  }  } |
| public void SetIoValue( string readPlcdata )  {  **// 문자열을 파싱한다. 일단 / 로 구분**  var seps = readPlcdata.Split("/"[0]);  if (seps.Length < 5) return;  for (int i = 0; i < 4; ++i)  {  **// 문자열을 , 를 짤라서 파싱한다.**  var item = seps[i+1].Split(","[0]);    var strInputOutput = item[0];  var indexStr = int.Parse(item[1]);  **// 해당하는 접점에 데이타를 저장한다.**  IoType typeIo = FindIo(IoType.Input, i, 0) != null ? IoType.Input : IoType.Output;    for (int j = 0; j < 15; ++j)  {  var valueInt = int.Parse(item[j+2]);  var inputio = FindIo(typeIo, i, j);    inputio.IoUnit.Value = (byte)valueInt;  }  }  } |

1. **INSPECT 기능** 
   1. 장치오브젝트들을 배치하는 Scene모드에서 각 장치의 각종 설정을 할 수 있는 Inspect 창이 제공된다.
      1. 위치, 회전, 이름 설정 등을 공통으로 할 수 있는 **공통제어 콤포넌트**가 있고.
      2. 접점의 연결을 할 수 있는 **접점연결 제어 콤포넌트**, **파워 설정 콤포넌트**등도 있다.
      3. INSPECT창 밑에 모든 콤포넌트들이 들어있으며, 설정을 할 수 없는 콤포넌트들을 Active FALSE를 주어서 자동으으로 설정할 수 있는 콤포넌트들만 노출한다.
   2. 인스펙트창의 유니티 구성
      1. 
      2. VerticalLayerGroup을 사용하여 하위에 있는 Inspect콤포넌트들을 자동으로 정렬을 해준다. 줄을 맞추어서 오른쪽 창에 보여주게 되며 숨기게 되고 자동으로 보이는 것만 정렬해서 보여준다.
   3. 인스펙트 창의 기본구조
      1. 인스펙트 창은 UiElement 를 상속 받아서 구현하는데, 크게 다음과 같은 구성이 존재한다.
         1. 제목설정
         2. 장치와의 연결, 유효성검사 (유효성에 따라서 숨김)
         3. 장치의 데이터가 바뀔 때 호출되어 인스펙트 창에 반영.
         4. 인스펙트 창에 의해서 장치의 데이터 설정.
         5. 기본 베이스 클래스 UiElement 창을 살펴보면 다음과 같다.

|  |
| --- |
| public class UiElement : MonoBehaviour  {  public Text textTitle;    **// 장치 오브젝트의 연결**  public virtual EquipmentEnuit data { get; set; }  public virtual EquipmentEnuit Data  {  get { return data; }  set { data = value; UpdateData(); UpdateChange();  Activity = IsValid();  }  }  **//장치 오브젝트의 유효성 검사**  public virtual bool IsValid()  {  return data != null;  }  **//장치오브젝트의 활성화/비활성화**  public bool Activity  {  get { return gameObject.activeSelf; }  set { gameObject.gameObject.SetActive(value); }  }  **//항상 호출되는 update 함수**  public virtual void UpdateData()  {  }  **//값이 변경할때만 호출되는 이벤트성 함수**  **//이때 장치오브젝트의 값을 인스펙트 콤포넌트에 업데이트한다.**  public virtual void UpdateChange()  {  }  **//Update를 구현하지 않고 따로 일괄적으로 콜백함수를 관리하는 TimeCall함수에 Update함수를 등록하여 사용한다. 퍼포먼스 향상을 위함.**  public virtual void Start()  {  TimeCall.Instance.AddAlwaysCall(delegate () { UpdateData(); });  }  } |
| **// 공통적으로 사용하는 위치포지션 설정 인스펙트 콤포넌트**  public class UiElementVector3 : UiElement  {  **// 현재 위치를 반영하여 UI표시**  public Text textPosition;  **//각각 x,y,z 값을 입력받기 위한 UI**  public InputField input1;  public InputField input2;  public InputField input3;  public UIButton btn;    Vector3 prevPosition { get; set; }    public override void UpdateData()  {  if (data == null) return;  **//현재 위치를 반영**  var position = data.transform.localPosition;  textPosition.text = string.Format("{0:f3}, {1:f3}, {2:f3}", position.x, position.y, position.z );    **// 뭔가 움직이면 입력 창도 바꾸어 준다.**  if (prevPosition != position)  UpdateChange();    prevPosition = position;  }  public override void UpdateChange()  {  if (data == null) return;  **// 입력창에도 변수업데이트, 값이 변화 할때만 업데이트 해준다.**  **// 값이 변하지 않을경우는(움직이지 않을경우) 입력창에서 입력하여 장치개체의 위치를**  **// 바꿔줄수 있다.**  var position = data.transform.localPosition;  input1.text = string.Format("{0:f3}", position.x);  input2.text = string.Format("{0:f3}", position.y);  input3.text = string.Format("{0:f3}", position.z);    Data.UpdateComponents();  }  public override void Start()  {  base.Start();  **// set 버튼을 누를경우 현재 위치를 입력창에서 입력된 값으로 변경**  btn.callbackDown = delegate( UIButton btn )  {  btn.SpellComponent.ScaleEffect();    var vt3 = new Vector3(  float.Parse(input1.text),  float.Parse(input2.text),  float.Parse(input3.text));    data.transform.localPosition = vt3;  UpdateChange();  };  }  } |
| **// 실린더의 스피드를설정하는 인스펙트 콤포넌트**  public class UiElementCylinderSpeed : UiElement  {  public Slider slider;  public Text textValue;  MoveComponent translater;    public override void UpdateData()  {  }  public override void UpdateChange()  {  if (data == null) return;  **//장치오브젝트의 MoveComponent를 얻어와서 실제로 작동하는 speed값에 반영한다.**  translater = data.GetComponentInChildren<MoveComponent>();    slider.value = translater.speed \* 20.0f;  slider.onValueChanged.Invoke(slider.value);  }  public override void Start()  {  TimeCall.Instance.AddAlwaysCall(delegate () { UpdateData(); });    slider.onValueChanged.AddListener  (  delegate (float value)  {  Debug.Log(" OnChange==>" + value);  textValue.text = $"{value:f2}";  translater.speed = value \* 0.05f;  }  );  UpdateChange();  }  public void SliderValueChange( float value )  {    }  } |

* + 1. **PLC와의 관계가 필요한 경우** 
       1. PLC와의 점점 연결 등을 인스펙트창을 통해서 바로 할 경우 **IOComponent의 장치 연결설정이 필요**하므로 Component를 기본 참조로 하고 **추가적인 PLC의 연결구성을 읽어오는** 인스펙트 콤포넌트가 필요하다.
       2. 이런 경우에 필요한 **UiElementIo 베이스클래스**를 정의 한다. 그리고 접점 연결을 하는 **UiElementISlotSelect** 클래스도 살펴보자.

|  |
| --- |
| public class UiElementIo : MonoBehaviour  {  public Text textTitle;  **//PLC의 참조도 만들어둔다.**  public virtual PlcEnuit plc { get; set; }  public virtual PlcEnuit Plc  {  get { return plc; }  set { plc = value;  UpdatePLC(); }  }  **// 이 인스펙터는 IOComponent값을 기본값으로 설정한다.**  public virtual IOComponent data { get; set; }  public virtual IOComponent Data  {  get { return data; }  set { data = value;    UpdateData();  UpdateChange();  Activity = IsValid();  }  }  public virtual bool IsValid()  {  return data != null;  }  public bool Activity  {  get { return gameObject.activeSelf; }  set { gameObject.gameObject.SetActive(value); }  }  public virtual void UpdateData()  {  }  public virtual void UpdateChange()  {  }  public virtual void UpdatePLC()  {  }  public virtual void Start()  { TimeCall.Instance.AddAlwaysCall(delegate () { UpdateData(); });  }  } |
| public class UiElementISlotSelect : UiElementIo  {  public Dropdown dropdown;    public List<DataContainer<string, EquipmentEnuit.IoElement>> DropdownInfo { get; private set; }  public override void UpdateData()  {  }  **// 무언가 변경될때 호출됨 현제 오브젝트의 이름을 설정**  public override void UpdateChange()  {  if (data == null) return;  SetName();  }  **// PLC의 정보가 바뀌어서 이를 인스펙트콤포넌트에 반영할떄**  public override void UpdatePLC()  {  base.UpdatePLC();    var selectIndex = 0;  NodeEunit connectNode = (Data == null) ? null : (Data.Node == null) ? null : Data.Node.ConnectNode;  var selectText = dropdown.options.Count < 1 ? string.Empty : dropdown.options[dropdown.value].text;    **// PLC 에서 접점 모든접점 정보를얻어온다.**  DropdownInfo = Plc.GetStringIos();  **// dropdown ui 개체를 초기화한다.**  dropdown.ClearOptions();    **// 다른 타입들은 제거한다.**  DropdownInfo.RemoveAll(data => (data.Second != null) && ( data.Second.Kind != Data.Type) );  **//현재 선택된 접점을 반영하여 드롭다운에 동일한 값이 선택된 것으로 설정한다..**  for (int i = 0; i < DropdownInfo.Count; ++i)  {  var item = new Dropdown.OptionData();  var io = DropdownInfo[i].Second;  item.text = DropdownInfo[i].First;  **// 현재선택된값이라면 이걸 현재설정된 값으로 반영.**  if(io != null && connectNode != null )  {  if (io.Kind == connectNode.Kind &&  io.KindIndex == connectNode.KindIndex &&  io.Index == connectNode.Index )  selectIndex = i;  }    **// dropdown ui에 dropdown 항목으로 추가.**  dropdown.options.Add(item);  }    **// 연결이 끊켰을때.. 무조건 0으로 0은 not selected,**  **// 1로 한번도 설정하는 것은 dropdown내부적으로 같은값일경우는 update를하지 않기 때문에**  **// 무조건 설정하는 걸로 하기 위함이다.**  if (connectNode == null)  {  dropdown.value = 1;  dropdown.value = 0;  return;  }    if (dropdown.value == selectIndex && selectIndex != 0)  dropdown.value = 0;  else  if (dropdown.value == selectIndex && selectIndex == 0)  dropdown.value = 1;    dropdown.value = selectIndex;    }  **// 현재 접점의 이름을 보여주기 위하여 호출된다. 이 이름은 각각의 장치마다 다르기 때문에 이 다른 이름들을 정의 하는 문자열 테이블도 사용된다. 이 문자열 테이블은 dicComponentName에 정의되어있다.**  public void SetName()  {  textTitle.text = UiElementIo.dicComponentName[Data.Node.Equipment.Type][Data.Type][Data.Index];  }  public virtual void Start()  {  SetName();  TimeCall.Instance.AddAlwaysCall(delegate () { UpdateData(); });    dropdown.onValueChanged.AddListener  (  delegate (int select)  {  if( select == 0 )  {  Data.Node.DestroyConnect();  return;  }  if (DropdownInfo[select].Second != null)  {  // 실제 노드 연결을 한다.  Data.Node.MakeConnect(DropdownInfo[select].Second.NodeUnit);  }    }  );  }  } |
|  |

1. **작동 콤포넌트**
   1. PLC로 부터 오는 값에 의해서 작동되는 콤포넌트들은 몇 가지가 있다. 이들은 장치들의 작동을 씬뷰에서 보여주어 시뮬레이션 역할을 보여준다. 그 콤포넌트들은 다음과 같은 콤포넌트들이 있다.
      1. **OnoffComponent** 위에서 설명한 값에 따라서 특정개체를 켜주거나 꺼주거나 한다. 특정개체 밑이 하위 개체가 다 꺼지므로 상위 개체는 형식적인 표시를 하고 하위로 여러 가지 개체들을 넣어주는 것도 좋은 방법이다. 이 개체는 위에서 설명.
      2. **MoveComponent 는** 특정한 값에 따라 개체를 움직이는 역할을 한다. 움직이는 최종위치는 0과 1에 따라 두 가지 장소가 존대 한다. 이는 Transform 으로 참조하기 때문에. Position으로 Lerp하는 방법으로 위치를 변경한다.

|  |
| --- |
| MoveComponent.cs |
| public class MoveComponent : BaseComponent  {  **// 값이 1일때 움직일 곳**  public Transform transOn;  **//값이 0일때 움직일 곳**  public Transform transOff;  **//움직일 대상**  public Transform transMove;  **// 스피드**  public float speed;  **// 참조하는 IOComponent**  public IOComponent ioIn;  **//대상 끝까지 다 갔을 때 값을 넘겨주는 input Component**  public IOComponent ioInBk;  public IOComponent ioOut;  void Start()  {  Handle = ioOut.Value;  }  void Update()  {  var isSelected = Equipment.IsOutlined;    if (isSelected == true) **// 키보드로 강제로 처리하는 부분 테스트 목적으로 추가함.**  { if (Input.GetKey(KeyCode.Alpha1) == true)  ioOut.Value = 1;  else  ioOut.Value = 0;  }  **// Handle 에 의해서 Lerp 는 방식으로 이동을 처리한다. 따라서 Handle 은 0~1 의 값**  Handle += ioOut.Value == 1 ? Time.deltaTime \* speed : Time.deltaTime \* -speed;  Handle = Mathf.Clamp(Handle, 0.0f, 1.0f);    **// 실제위치이동**  MoveEffect(Handle);  ioIn.Value = (Handle == 1.0f) ? (byte)1 : (byte)0;  ioInBk.Value = (Handle == 0.0f) ? (byte)1 : (byte)0;  }  **// 실제 이동은 Vector3.Lerp 로 처음 값과 마지막값중에 Handle값으로 고르는 방법.**  void MoveEffect( float value )  {  transMove.transform.localPosition = Vector3.Lerp(transOff.localPosition,transOn.localPosition, value);  }} |

* + 1. **RotateComponent**는 MoveComponent와 동일하지만 rotation을 한다.

|  |
| --- |
| RotateComponent.cs |
| void MoveEffect( float value )  {  transMove.transform.localRotation = Quaternion.Lerp(transOff.localRotation, transOn.localRotation, value);  } |

* + 1. **CollisionComponent**는 특정영역에 특정 레이어의 오브젝트를 검출 할 때의 콤포넌트이다. Product 를 검출할 때 사용되므로 OnTrigger에서 충돌될 때 MovableProduct 스크립트를 찾는다. 이것이 있을 때만 충돌체크를 한다.
       1. 여기에서 충돌 검출 방법은 유니티에서 콜라이더에 Trigger를 채크할때 발생하는 OnTriggerEnter/Exit 함수로 충돌처리를 하게 된다.
       2. OnTrigger에 충돌이 검출되려면 어느 한쪽이 **RigidBody**가 존재해야 하는데, 주로 움직이는 개체는 MovableProduct 이므로 Product 쪽에만 Rigidbody 가 있다.
       3. CollisitionComponent는 한가지 특수한 행동을 하게 하는데, **Power를 선택하는 기능**이다. 이는 Sensor가 두 가지 파워를 제공하는 독특한 구조를 갖기 때문에 이 제공되는 두 가지 Power중 어느 것을 선택할지를 결정하는 역할을 한다. 나중에 인스펙터 **UiElementSensorType** 에서 이 값을 바꾼다. 어떤 파워를 고르게 되는 것은 두 가지 선택지가 있다. 0v로 연결되는 파워 라인과 24v로 연결되는 파워 라인.
       4. Sensor의 특성이 또한 검출하는 종류의 특성을 갖게 되는데, 이 검출하는 종류는 검은색이냐, 가공되었느냐 등을 추가로 판단하게 된다. 이 판단에 대한 유무와 그 판단 옵션들이 Product에 이미 존재하므로 이것들과 서로 동일한지를 판단하게 된다.
       5. 이 충돌검사는 그리고 **복수개가 될 수 있다**. 여러 개의 충돌이 되면, 하나의 충돌 체가 빠져나가더라도 아직 여러 개가 충돌되어 있기 때문에 단순한 enter/exit 로는 판별을 다 할 수가 없고 그래서 충돌 **카운터(StackCounter)**라를 것을 이용하여 복수개의 충돌을 검출하고 있다.

|  |
| --- |
| CollisionComponent.cs |
| public class CollisionComponent : BaseComponent  {  **// 검출하는 product의 레이어**  public LayerMask layer;  **//참조하는 IOComponent**  public IOComponent io;  **//Sensor 의 특성을 구분하는구별자,Seonsr 에 따라서 Power를 다르게 선택되기 때문에**  **//이 값에 의해서 센서가 어떤 파워를 고를지 결정한다.**  public int PowerSelect = 0;    public bool checkOption;  public bool isBlackCheck;  public bool isModifyedCheck;  public bool IsCollision { get; set; }    BoxCollider collider { get; set; }  MoveableProduct product;  int StackCounter { get; set; }    // Start is called before the first frame update  void Start()  {  collider = GetComponent<BoxCollider>();  collider.isTrigger = true;  }  void Update()  {  **// 검출 옵션, 검은색이냐 가공되었느냐 등을 판단.**  if (checkOption == true && product != null)  {  if( isBlackCheck == product.IsBlack &&  isModifyedCheck == product.IsModifyed &&  IsCollision == true  )  io.Value = 1;  else  io.Value = 0;    **// 파워에 대한 선택인자 값. KindIndex를 여기서 선별해서 설정해 준다.**  **// 이 파워선별은 별도의 인스펙트창에서 설정에 의해서 변경된다.**  io.Node.KindIndex = io.Value == 1 ? PowerSelect : -1;  }  else  {  io.Value = IsCollision == true ? (byte)1 : (byte)0;  io.Node.KindIndex = IsCollision == true ? PowerSelect : -1;  }    }  **//충돌이 검출되면 호출**  **//StackCount를 사용하여 복수개가 충돌됨을 체크**  private void OnTriggerEnter(Collider other)  {  var isLayer = layer.Contains(other.gameObject.layer);    if (isLayer == true)  {  IsCollision = true;    **// Product에 존재하게 되는 MoveableProduct를 찾는다.**  product = other.gameObject.GetComponent<MoveableProduct>();  ++StackCounter;  }  }  private void OnTriggerExit(Collider other)  {  var isLayer = layer.Contains(other.gameObject.layer);    if (isLayer == true)  {  --StackCounter;    if (StackCounter < 1)  { IsCollision = false;  product = null;  }  }  }  public override void OnChangeTranslate()  {  }} |

* + 1. **AnimatorComponent** 콤포넌트는 IOComponet 의 값에 의해서 특정 애니메이션 기능을 수행하는 콤포넌트이다.
       1. 애니메이션을 수행하는 Animator가 추가되어 있어야 하고 이 Animator에는 Idle과 Action 의 두 가지 액션이 정의되어 있어야 한다.
       2. 이 콤포넌트는 복잡한 움직임이나 정확한 입출력 정보를 산출해 낼 필요가 없을 때 주로 사용된다. 값에 대해서 접점에 값을 보고할 필요가 없는 경우.

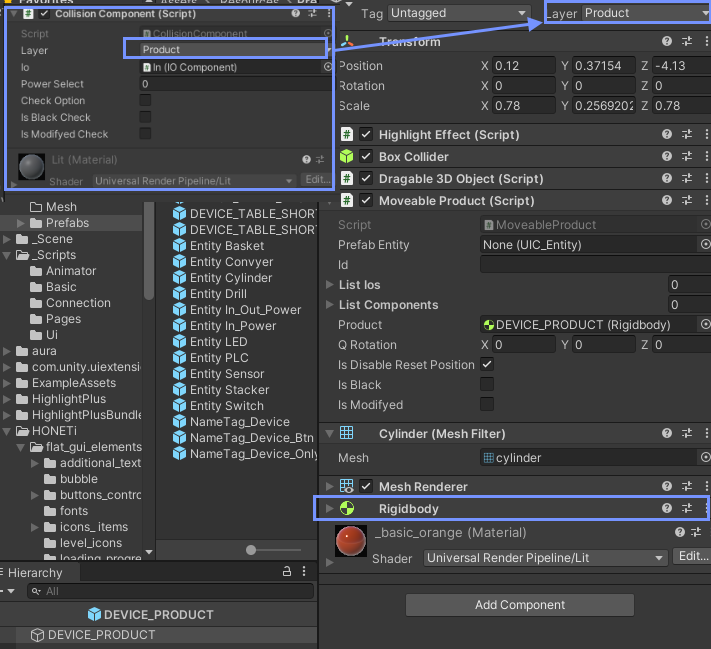
|  |
| --- |
|  |
| public class AnimatorComponent : BaseComponent  {  public IOComponent io;  public BaseComponent handleComponent;    Animator anima;  bool prevState { get; set; } = false;    private void Awake()  {  anima = GetComponentInChildren<Animator>();  }  // Start is called before the first frame update  void Start()  {  if( io != null )  Handle = io.Value;  }  void Update()  {  var isSelected = Equipment.IsOutlined;    // IOComponent 로 작동시킬경우  if ( io != null )  {  **// 값이 1일때 Action 애니메이션 수행**  if (prevState == false && io.Value == 1)  anima.Play("Action", -1, 0.0f);    **// 값이 0일때 Idle 애니메이션 수행**  if (prevState == true && io.Value == 0)  anima.Play("Idle", -1, 0.0f);    prevState = io.Value == 1 ? true : false;  }  Else  **//특정장치의Handle값에 의해서 작동할 경우, 이럴 경우는 input,output 값과 무관하게 작동**  if(handleComponent != null )  {  var HandleValue = handleComponent.Handle;    if (prevState == false && HandleValue == 1.0f)  anima.Play("Action", -1, 0.0f);    if (prevState == true && HandleValue < 1.0f)  anima.Play("Idle", -1, 0.0f);    prevState = (HandleValue == 1.0f ) ? true : false;  }  }  } |

* + 1. **ButtonComponent**는 추가된 UI의 Button의 값이 1이냐 0이냐에 따라서 IOComponent의 값을 설정하는 콤포넌트로 위에서 설명했다. 주로 스위치 같은 역할로 버튼을 누루거나 해서 값을 PLC로 보낼 때 사용한다.
    2. **CollisionStick** 콤포넌트는 특정개체가 충돌될 때 충돌 체를 원하는 위치에 고정시키는 역할을 한다. Snap기능과 비슷한 역할을 하는 콤포넌트이다.

|  |
| --- |
|  |
| public class CollisionStick : BaseComponent  {  public LayerMask layer;  public bool isRigidbody;  public IOComponent io;  public Transform stickPosition;  public float speed;  BoxCollider collider { get; set; }  List<Rigidbody> listMoveables;    void Start()  {  collider = GetComponent<BoxCollider>();  listMoveables = new List<Rigidbody>();  collider.isTrigger = true;  }  void Update()  {  if (io != null)  io.Value = (listMoveables.Count > 0) == true ? (byte)1 : (byte)0;  }  private void OnTriggerEnter(Collider other)  {  Debug.Log("Stick Trigger On==>" + collider.name + "/" + listMoveables.Count);  **// 충돌을 검출해내고 Rigidbody가 있는 경우만..**  **// Rigidibody 가 있는 경우가 Prodct일경 우 .**  var iscollision = layer.Contains(other.gameObject.layer);  if (iscollision == false) return;    **// 이미 관리대상이면 제외**  var rigid = other.GetComponent<Rigidbody>();  if (listMoveables.Contains(rigid) == true) return;    **// rigidbody 일경 우 고정형태 관리대상으로 추가한다. 고정형태의 경우는 rigidbody를 잡아서 드래그할 경우 고정형태가 되므로, 선택적인 개체를 선정한다.**  if (rigid.isKinematic == true )  listMoveables.Add(rigid);    Debug.Log("Stick Trigger On2==>" + collider.name +"/"+ listMoveables.Count);  }  private void OnTriggerExit(Collider other) { }  public override void OnChangeTranslate() { }  public Dragable3DObject GetDragableObject( GameObject gObject )  {  var dragable = gObject.GetComponentInParent<Dragable3DObject>();  if (dragable == null)  dragable = gObject.GetComponent<Dragable3DObject>();    return dragable;  }  private void LateUpdate()  {  List<Rigidbody> removes = new List<Rigidbody>();    **//관리대상 개체들을 위치를 고정시켜준다.**  for( int i=0;i< listMoveables.Count; ++i )  {  var moveOne = listMoveables[i];  var dragable = GetDragableObject(moveOne.gameObject);    Debug.Log("==>Stick Lates==>" + moveOne.name + "/" + listMoveables.Count);    **//원하는 위치로 고정**  moveOne.transform.position = stickPosition.position;    if (moveOne.isKinematic == false)  {  Debug.Log("==>Stick Not Kinemitic==>" + moveOne.name);  removes.Add(moveOne);  }  else  {  **// 마우스로 픽킹한 위치와 너무 떨어지게되면 관리 대상에서 풀어준다. 그럼 고정위치에서 풀리게된다.**  var distance = (moveOne.transform.position - dragable.LastGroundPosition).magnitude;    **//거리가 일정거리 이상이되면 풀어준다.**  if (distance > 3.4f)  {  Debug.Log("==>Stick Remove==>" + moveOne.name +"/" + distance);  removes.Add(moveOne);    **//다시 움직일수 있는 신호를 건내준다.**  var moveProduct = dragable.GetComponent<MoveableProduct>();  moveProduct.OnDragStart();  }  }  }  for (int i = 0; i < removes.Count; ++i)  listMoveables.RemoveAll( data=>data == removes[i]);  }    } |

* + 1. **CollisionTranslate** 콤포넌트는 **rigidbody** 오브젝트가 검출될 경우 이 오브젝트들을 지속적으로 이동시켜준다. 충돌영역에서 벗어나면 적용하지 않는다.
       1. 주로 Product등을 움직여주는 콘베이어벨트 등에서 사용한다.
       2. **Rigidbody의 velocity 값의 설정**으로 오브젝트를 움직이게 한다.

|  |
| --- |
|  |
| public class CollisionTranslate : BaseComponent  {  public LayerMask layer;  public Material uvMaterial;  public Vector2 uvDirection;  public bool isRigidbody;  public IOComponent io;  public Transform direction;  public float speed;  BoxCollider collider { get; set; }  **// 관리하는 rigidbodys들**  List<Rigidbody> listMoveables;    // Start is called before the first frame update  void Start()  {  collider = GetComponent<BoxCollider>();  collider.isTrigger = true;    listMoveables = new List<Rigidbody>();  }  void Update()  {  **// uv 애니메이션을 통해서 콘베이어밸트가 흘러가는 등의 모습을 보여준다.**  if( io.Value == 1 || Equipment.IsPower == true)  { if (uvMaterial != null)  uvMaterial.mainTextureOffset += uvDirection;  }    if (io.Node == null) return;  if (Equipment.IsPower == false) return;  if (io.Value == 0) return;    CheckBoxCollision();  //if (io != null)  // io.Value = CheckBoxCollision() == true ? (byte)1 : (byte)0;    }  public override void OnChangeTranslate()  {  }  private void OnTriggerEnter(Collider other)  {  var iscollision = layer.Contains(other.gameObject.layer);  if (iscollision == false) return;    **//이미 관리대상이 아닐경우만 추가해준다.**  var rigid = other.GetComponent<Rigidbody>();  if (listMoveables.Contains(rigid) == true) return;    listMoveables.Add(rigid);    //Debug.Log("===TriggerEnter====" + other.name +"/" + listMoveables.Count);  }  private void OnTriggerExit(Collider other)  {  var rigid = other.GetComponent<Rigidbody>();  listMoveables.RemoveAll(data => data == rigid);    //Debug.Log("===TriggerExit====" + other.name);  }  public bool CheckBoxCollision()  {  **// 관리 대상오브젝트들의 velocity를 설정해 줌으로서 움직이게 해준다.**  for( int i=0;i< listMoveables.Count; ++i )  listMoveables[i].velocity = direction.forward \* speed;    return listMoveables.Count > 0;  }} |

1. **PRODUCT**
   1. Product는 이 장치들간에서 생산되는 생산물. 장치들이 이 product에 가공을 하는 과정을 보여주는 장치들의 조합으로 볼 수 있다.
   2. 이 Product는 물리적으로 움직이기 때문에 **Rigidbody**가 포함되어있다.
   3. Product는 특정한 레이어를 가지며 이 레이어로 충돌 검출을 할 때 검출 대상으로 구분할 수가 있다. (Collision Component 등에서 검출)
      1. 
   4. Dragable3Dobject로 움직일 때 물리적용 받지 않는 상태로 움직이게 한다. 또한 y축으로 위로 끌어올려서 어떤 오브젝트 위에도 올려놓을 수 있게 한다.

|  |
| --- |
| * + 1. MoveableProduct.cs |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  public class MoveableProduct : EquipmentEnuit  {  public Rigidbody product;  public Quaternion qRotation;  public bool isDisableResetPosition;  **// 이 Product의 옵션들 검은색인가? 가공이 되어 있는가?**  public bool IsBlack;  public bool IsModifyed;  public void SetModifyed( bool ismodifyed)  {  IsModifyed = ismodifyed;  }  override protected void Awake()  {  }  override protected void Create()  {  **// 기계장치가 각종 작업의 대상인 생산물을 의미한다.**  **// 다른 장치가 하는 연결 노드 구성같은건 안 한다.**  Type = EunitType.Product;  qRotation = product.transform.localRotation;  }  override protected void UpdateUnit()  {  base.UpdateUnit();  }  **// 마우스로 자유로이 움직이고, 물리 개체는 iskinect 처리를 하여 같이 따라 다닐 수 있게 한다.**  **// 드래그 시작 마우스 버튼 누르는 시점**  public override void OnDragStart()  {  base.OnDragStart();  product.isKinematic = true;  product.transform.localRotation = qRotation;  if (isDisableResetPosition == false)  product.transform.localPosition = Vector3.zero;  }  **// 드래그 띠는 시점, 마우스 버튼 떼는 시점**  public override void OnDragEnd()  {  base.OnDragStart();  product.isKinematic = false;  }  } |

* 1. Product는 검출 시 속성에 의해서 속성체크를 할 수가 있는데 검은색 구분, 가공 구분 등 두 가지로 구분을 하여 충돌검출을 한다. MoveableProduct.cs 에 두 가지 항목이 있는 것을 확인할 수 있다. 또한 이 충돌검출은 CollisionComponent.cs 에서 하며 속성값을 체크해서 충돌검출을 한다.

|  |
| --- |
| 1. public class MoveableProduct : EquipmentEnuit 2. { 3. public Rigidbody product; 4. public Quaternion qRotation; 5. public bool isDisableResetPosition; 6. **public bool IsBlack;** 7. **public bool IsModifyed;** |
| void Update()  {  if (checkOption == true && product != null)  {  if( isBlackCheck == product.IsBlack &&  isModifyedCheck == product.IsModifyed &&  IsCollision == true  )  io.Value = 1;  else  io.Value = 0;    io.Node.KindIndex = io.Value == 1 ? PowerSelect : -1;  }  else  {  io.Value = IsCollision == true ? (byte)1 : (byte)0;  io.Node.KindIndex = IsCollision == true ? PowerSelect : -1;  }  } |
|  |

* 1. 이 CollisionComponent 의 옵션을 바로 체크해서 Inspect 창에서 구분하여 보여주는 UiElementSensorKind가 있다. 세가지 상태 OnlyCollision, Color, Modify 등 의 옵션을 제공한다.

|  |
| --- |
| 1. public class UiElementSensorKind : UiElement 2. { 3. public Dropdown dropdown; 4. CollisionComponent translater; 5. SensorBase unit; 6. public override bool IsValid() 7. { 8. if( data == null ) return false; 10. var unit = data as SensorBase; 11. return unit.senorKind != SensorType.OnlyCollision; 12. } 13. public override void UpdateData() 14. { 15. } 16. public override void UpdateChange() 17. { 18. if (data == null) return; 19. unit = data as SensorBase; 20. translater = data.GetComponentInChildren<CollisionComponent>(); 21. dropdown.ClearOptions(); 22. if(**unit.senorKind** == SensorType.OnlyCollision) 23. { 24. var item = new Dropdown.OptionData(); 25. item.text = "충돌만"; 26. dropdown.options.Add(item); 27. item = new Dropdown.OptionData(); 28. item.text = "충돌만"; 29. dropdown.options.Add(item); 30. translater.checkOption = false; 31. translater.isBlackCheck = false; 32. translater.isModifyedCheck = false; 33. dropdown.value = 1- dropdown.value; 34. dropdown.value = 0; 36. } 37. if (**unit.senorKind** == SensorType.Color) 38. { 39. var item = new Dropdown.OptionData(); 40. item.text = "검정색"; 41. dropdown.options.Add(item); 42. item = new Dropdown.OptionData(); 43. item.text = "비검정색"; 44. dropdown.options.Add(item); 45. translater.checkOption = true; 46. translater.isModifyedCheck = false; 47. var oldValue = dropdown.value; 48. var newValue = (translater.isBlackCheck == true) ? 0 : 1; 49. if( newValue == oldValue ) 50. dropdown.value = 1-oldValue; 52. dropdown.value = newValue; 53. } 54. if (**unit.senorKind** == SensorType.Modify) 55. { 56. var item = new Dropdown.OptionData(); 57. item.text = "비가공"; 58. dropdown.options.Add(item); 59. item = new Dropdown.OptionData(); 60. item.text = "가공"; 61. dropdown.options.Add(item); 62. translater.checkOption = true; 63. translater.isBlackCheck = false; 64. var oldValue = dropdown.value; 65. var newValue = (translater.isModifyedCheck == true) ? 0 : 1; 66. if (newValue == oldValue) 67. dropdown.value = 1 - oldValue; 69. dropdown.value = newValue; 70. } 71. } 72. public override void Start() 73. { 74. TimeCall.Instance.AddAlwaysCall(delegate () { UpdateData(); }); 75. dropdown.onValueChanged.AddListener 76. ( 77. delegate (int select) 78. { 79. if (unit.senorKind == SensorType.OnlyCollision) 80. { 81. if (select == 0) 82. { translater.checkOption = false; 83. translater.isBlackCheck = false; 84. translater.isModifyedCheck = false; 85. } 86. } 87. if (unit.senorKind == SensorType.Color) 88. { 89. if (select == 0) 90. { translater.checkOption = true; 91. translater.isBlackCheck = true; 92. translater.isModifyedCheck = false; 93. } 94. if (select == 1) 95. { translater.checkOption = true; 96. translater.isBlackCheck = false; 97. translater.isModifyedCheck = false; 98. } 99. } 100. if (unit.senorKind == SensorType.Modify) 101. { 102. if (select == 0) 103. { translater.checkOption = true; 104. translater.isBlackCheck = false; 105. translater.isModifyedCheck = true; 106. } 107. if (select == 1) 108. { translater.checkOption = true; 109. translater.isBlackCheck = false; 110. translater.isModifyedCheck = false; 111. } 112. } 113. } 114. ); 115. if (data != null) 116. translater = data.GetComponentInChildren<CollisionComponent>(); 117. } 118. } |