MOVEMENT IN VIRTUAL REALITY

Paul **TĂMAȘ**, Andreea **POP**
Technical University of Cluj-Napoca, Department of Electrical, Electronic and Computer Engineering,
paultamas32@gmail.com, andreeapop05@yahoo.com

Keywords: Software implementation, Virtual Reality, Application programming interfaces

Abstract: *In this paper we talk about the user's movement in virtual reality. The programmable part of the application is made in the C ++ programming language, and as the engine for running the graphic image we used Unreal Engine developed by Epic Games. Fragments of code appear in the paper for the calculation and positioning of the user in space, on a two-dimensional plane. The work also contains instructions to be able to make the settings so that the application can be controlled by the glasses levers. The whole project is developed for HTC VIVE PRO virtual reality smart glasses.*

1. INTRODUCTION

The technology that facilitates the operation of complex systems, consuming information and turning it into knowledge (that most valuable of human resources), is the realm of virtual reality. This special issue of IEEE Computer Graphics and Applications brings together articles describing virtual reality technology and applications being pursued worldwide. We felt the time was right for a peer-reviewed special issue because the field has produced an enormous amount of hype. This can damage credibility and obscure the real industry achievements and the extraordinarily important work being done.

Although the terms cyberspace and virtual reality have been around for years, virtual reality as an industry is in its infancy. (Evans and Sutherland demonstrated the first head-mounted stereo display in 1965). The term “virtual reality” is credited to Jaron Lanier, founder of VPL Research: earlier experimenters, like Myron Krueger in the mid-l970s, used phrases like *‘artificial reality.” William Gibson coined “cyberspace” in his 1984 science fiction novel, Neuromancer. Few technologies in recent years have evoked such fiery discussions in the technical community, and fewer still have sparked such passionate involvement of the humanities and the cultural sector. Maybe the humanities community reacts because the VR interaction is so tightly coupled to the human senses. Perhaps the cultural sector clamors for a role in the evolution of VR because the technology is finally interfacing with the human, rather than the human interfacing with the technology. Whatever the reasons, VR is more a convergence of previously disparate disciplines than a whole new branch of technology. It simply takes a fresh look at human interaction. Evolving from user interface design, flight and visual simulation, and telepresence technologies, VR is unique in its emphasis on the experience of the human participant. VR focuses the user's attention on the experience while suspending disbelief about the method of creating it. We feel that neither the devices used nor the level of interactiveness or fidelity determine whether a system is "VR." The quality of the experience is crucial. To stimulate creativity and productivity, the virtual experience must be credible. The "reality" must both react to the human participants in physically and perceptually appropriate ways, and conform to their personal cognitive representations of the microworld in which they are engrossed. The experience does not necessarily have to be realistic-just consistent. Articles by Stephen Ellis and by John Latta and David Oberg consider the frequently forgotten human side of VR systems.

Virtual reality today bears a striking resemblance to the early stages of computer graphics in the mid-1960s to the early 1970s. The products seem to be "a solution in search of a problem." As with early computer graphics products, the entry-level costs are relatively prohibitive. A complete VR environment, including workstations, goggles, body suits, and software, is in the range of $50,000 to $100.000. In an attempt to suggest low-cost methods, a new magazine called PCVR: Virtual Reality and the IBM Personal Compiiter (Gradecki Publishing, Laramie, Wyoming) publishes articles such as how to build a head-mounted display for under $500. The serious limitations of the technology give rise to a number of apologists. At one VR meeting in 1992, the general attitude was, "Although the pictures aren't very good, we really don't need great pictures to achieve our objectives. They don't have to be in real time, and they don't need to be terribly realistic.'' We see a strong analog to the early days of computer graphics, when all that was affordable was a fairly static, monochrome, storage tube display. The early rationalization was that we didn't need color or dynamics. Time has shown that once the technology became affordable, color and realism were much preferred. We believe the VR community will reach the same conclusion as the technology progresses. Much research into the various elements of VR technology remains to be done, for example, control and navigation metaphors for HMD point-of-view applications. We need to find a comfortable way for a user to move a POV while attempting to interact with objects and simultaneously control and gather information. Mark Bolas' article discusses these issues. To accommodate both "immersive/inclusive" experiences and multiple serial users, Bolas suggests the headcoupled boom-supported stereoscopic display.

Many application areas require this type of capability to integrate a VR system into the work environment. As the price comes down, these systems will surely see success in many fields.

2. CREATING MOTION CONTROLS

In order to be able to create the movement of the character through the camera, it is first necessary to take over the characteristics of the camera and to make the connection between the C ++ code and the Blueprint in which the character was created.

The camera features are taken over by the following UCameraComponent command (this command takes over the camera components), because it does not need to be visible in the entire application source, it will be private. The syntax for declaring this feature of the application is as follows:

private:
UPROPERTY(VisibleAnywhere) Classic
UCameraComponent The room;*

That *Camera creates a reference of the application components. The reference will be used in the program file, where the data processing will be done, so that the motion effect can be created. "UPROPERTY(VisibleAnywhere) ”is the method by which the component is assigned the visibility property throughout the program file, after inclusion. This property is only required for the latest Unreal Engine versions (Unreal Engine versions 4.15 and later). It must be put before each function that we implement in the header.

In the program file, we must first set the attachment, and the parent class will take the basic components by yourself.

In order to make it possible to move the character in the application workspace, we need to think of that plan as a two-dimensional (2D) axis system (*figure 1*). Suppose we want to make a forwardbackward motion, this involves moving on one of the axes of the plane, preferably the "Y" axis is chosen. In order to be able to perform movement, in C ++, we need a function derived from the parent class, which takes data from the forwarding vector (forwardVector) and transmits them to the class defined in the header. For the forward-backward movement we can do it by incrementing or decrementing by one or more pixels depending on the speed with which we want to move. Increment or decrement can only be done with an integer, i.e. it cannot be done in half pixels.

„For right-to-left or reverse movement, the procedure is quite similar. It involves the movement on one of the axes of the plane, different from the axis we chose for the front – rear motion, preferably choose the "X" axis. In order to be able to perform motion, in C ++, we need a function derived from the parent class, which takes data from the targeting vector (rightVector) and transmits it to the class defined in the header. As with the previous function, you cannot increment or decrement by half a pixel.

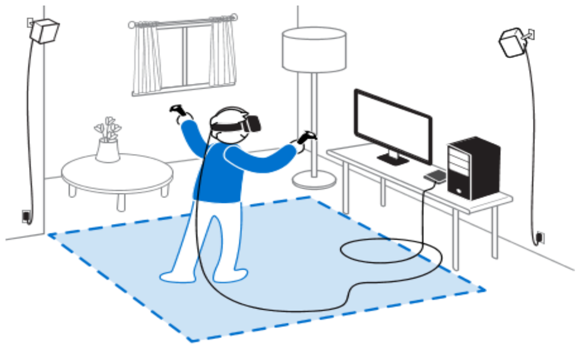
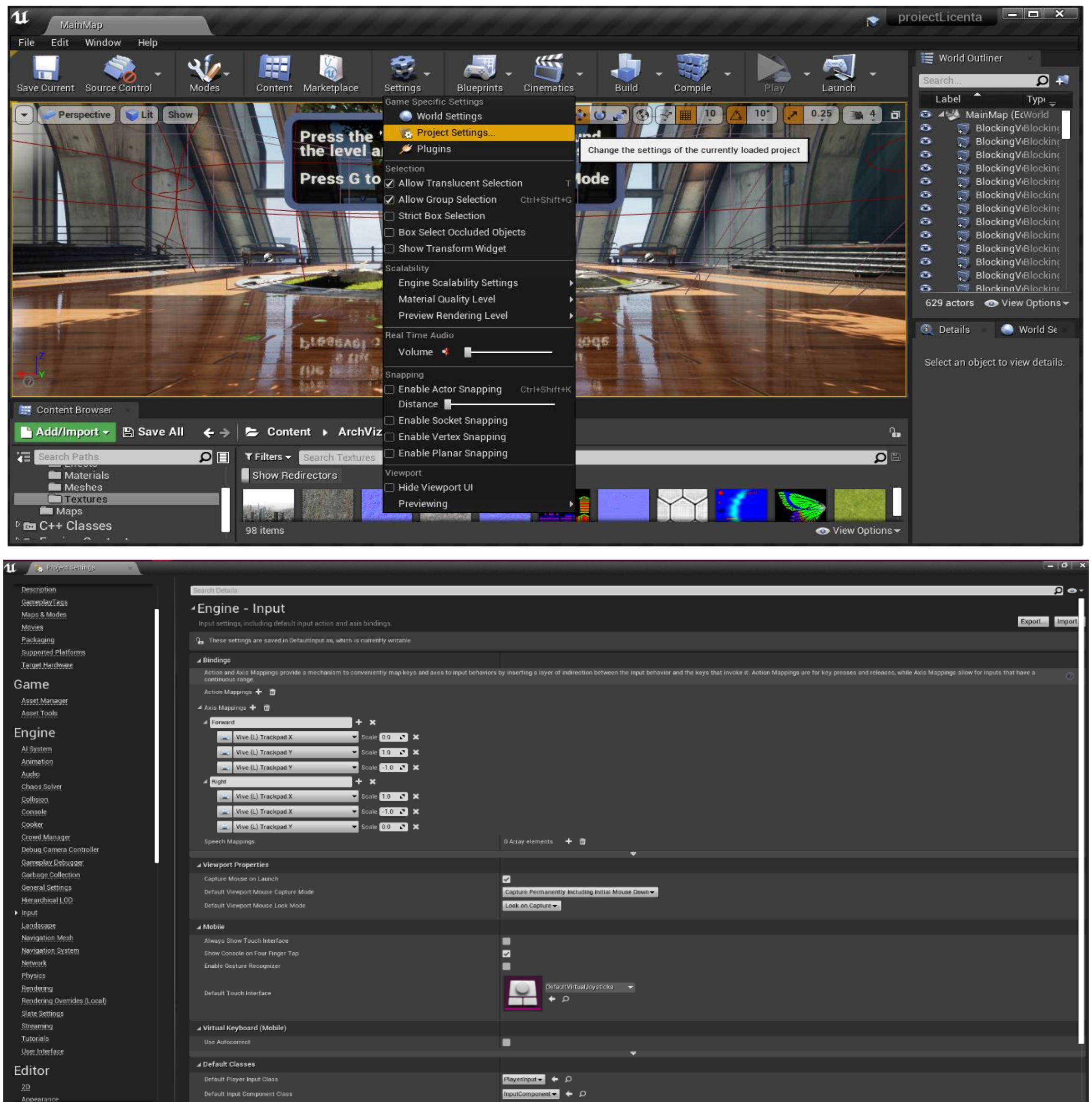


Fig. 1. Example of movement plan

For the physical controllers part, we will use the levers included in the HTC VIVE PRO virtual reality package. Setting them and setting specific buttons for movement is done by Unreal Engine: Settings -> Project settings -> Input -> Bindings. More precisely, it can be seen how the settings in the following set of figures are made.



Engine - Input

Input settings, including default input action and axis bindings.

These settings are saved in DefaultInput.ini, which is currently writable.

Bindings

Action and Axis Mappings provide a mechanism to conveniently map keys and axes to input behaviors by inserting a layer of indirection between the input behavior and the keys that invoke it. Action Mappings are for key presses and releases, while Axis Mappings allow for inputs that have a continuous range.

Action Mappings

Axis Mappings

Forward		
	Vive (L) Trackpad X	Scale 0.0
	Vive (L) Trackpad Y	Scale 1.0
	Vive (L) Trackpad Y	Scale -1.0
Right		
	Vive (L) Trackpad X	Scale 1.0
	Vive (L) Trackpad X	Scale -1.0
	Vive (L) Trackpad Y	Scale 0.0

Speech Mappings 0 Array elements

Viewport Properties

Capture Mouse on Launch	✓
Default Viewport Mouse Capture Mode	Capture Permanently Including Initial Mouse Down
Default Viewport Mouse Lock Mode	Lock on Capture

Mobile

Always Show Touch Interface	
Show Console on Four Finger Tap	✓
Enable Gesture Recognizer	
Default Touch Interface	DefaultVirtualJoysticks

Virtual Keyboard (Mobile)

Use Autocorrect	

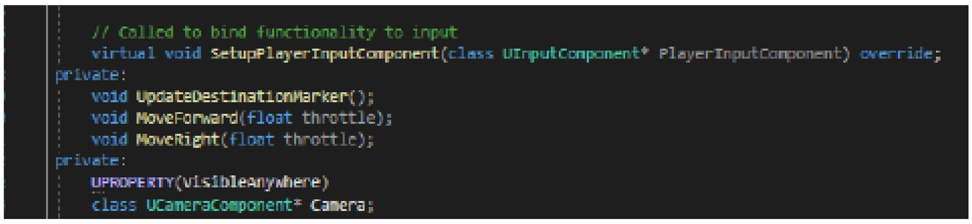
Default Classes

Default Player Input Class	PlayerInput
Default Input Component Class	InputComponent

Fig. 2. Lever controller setting

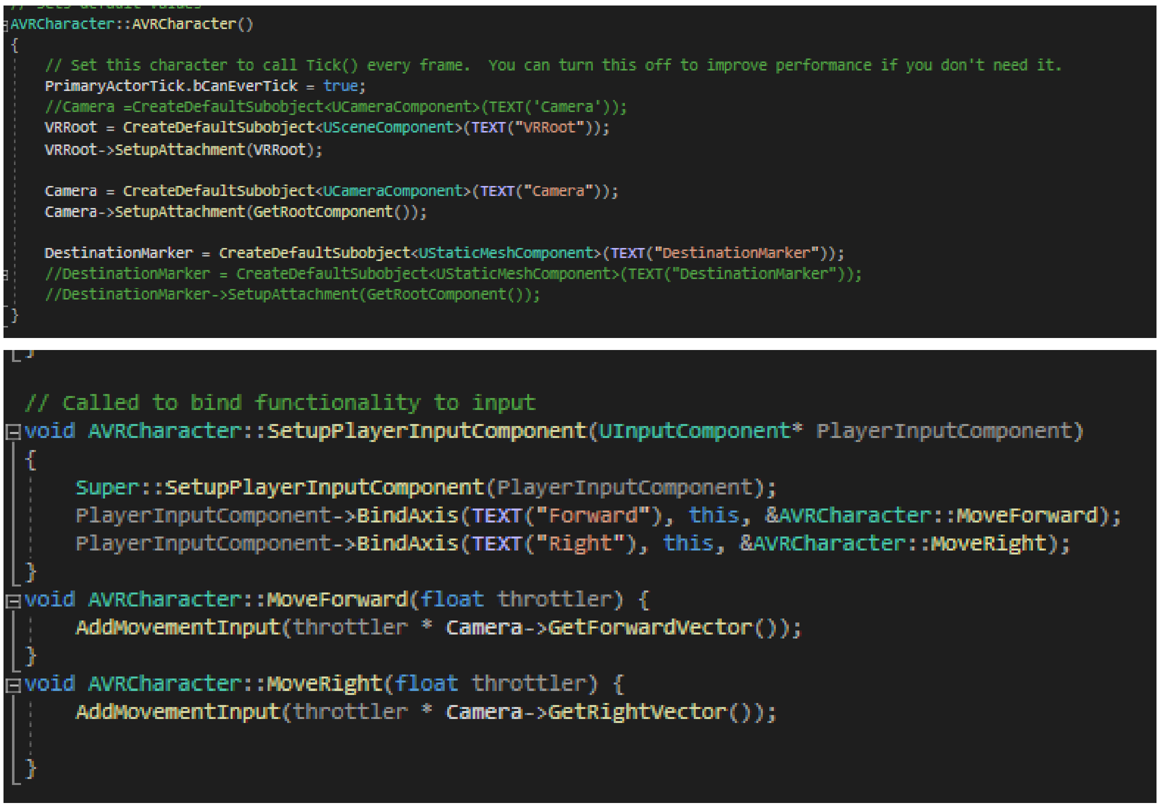
To add functions (axes) we click on "+" next to "axis mappings" and a place will appear where we can enter the name of the axis, this name must be identical to the reference we gave-o in C ++ code. From *figure 2* it can be seen that we named the axes "forward" and "right". For the "forward" axis I set the "Y" axis of the trackpad on the left joystick, and for "right" I set the "X" axis of the same trackpad. Since after several tests the movement was diagonal, I noticed that for each "forward" or "right" axis, the "X" or "Y" axis, which is not used for movement, must be set to 0.

The following set of figures exemplifies some of the code in C ++ for capturing controls on levers.



```
    // Called to bind functionality to input
    virtual void SetupPlayerInputComponent(class UInputComponent* PlayerInputComponent) override;
private:
    void UpdateDestinationMarker();
    void MoveForward(float throttle);
    void MoveRight(float throttle);
private:
    UPROPERTY(visibleAnywhere)
    class UCameraComponent* Camera;
```

Fig. 3. Header file



```
AVRCharacter::AVRCharacter()
{
    // Set this character to call Tick() every frame.  You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;
    //Camera =CreateDefaultSubobject<UCameraComponent>(TEXT('Camera'));
    VRRoot = CreateDefaultSubobject<USceneComponent>(TEXT("VRRoot"));
    VRRoot->SetupAttachment(VRRoot);

    Camera = CreateDefaultSubobject<UCameraComponent>(TEXT("Camera"));
    Camera->SetupAttachment(GetRootComponent());

    DestinationMarker = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("DestinationMarker"));
    //DestinationMarker = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("DestinationMarker"));
    //DestinationMarker->SetupAttachment(GetRootComponent());
}
```

```
// Called to bind functionality to input
void AVRCharacter::SetupPlayerInputComponent(UInputComponent* PlayerInputComponent)
{
    Super::SetupPlayerInputComponent(PlayerInputComponent);
    PlayerInputComponent->BindAxis(TEXT("Forward"), this, &AVRCharacter::MoveForward);
    PlayerInputComponent->BindAxis(TEXT("Right"), this, &AVRCharacter::MoveRight);
}
void AVRCharacter::MoveForward(float throttler) {
    AddMovementInput(throttler * Camera->GetForwardVector());
}
void AVRCharacter::MoveRight(float throttler) {
    AddMovementInput(throttler * Camera->GetRightVector());

}
```

Fig. 4. Program file

3. CONCLUSION

To develop an application that serves the field of architecture, you need a team of people who can create textures and objects, to be able to assemble them in order to come up with a useful application.

What we did in the app is the motion control part, the adjustment part collisions, introduction and modification of existing objects and the part of the user's interaction with objects in the application by inserting virtual hands using the controller.

REFERENCES

[1] M. A. Gigante, *Virtual reality: definitions, history and applications*, Virtual reality systems. Academic Press, pp. 3-14, 1993.

[2] J. M. D. Delgado et al, *A research agenda for augmented and virtual reality in architecture, engineering and construction*, Advanced Engineering Informatics, vol. 45, 101122, 2020.

[3] G. Schmitt, et al, *Toward virtual reality in architecture: concepts and scenarios from the architectural space laboratory*, Presence: Teleoperators & Virtual Environments, vol. 4, no. 3, pp. 267-285,1995.

[4] G. Reitmayr and D. Schmalstieg, *An open software architecture for virtual reality interaction*, Proceedings of the ACM symposium on Virtual reality software and technology, 2001.

[5] F. Agnello, F. Avella and S. Agnello, *Virtual reality for historical architecture,* International Archives of the Photogrammetry, Remote Sensing & Spatial Information Sciences, vol. XLII-2/W9, 2019.

[6] M. Paranandi and T. Sarawgi, *Virtual reality on architecture: enabling possibilities*, 2002.

[7] A. Sanders, *An introduction to unreal engine 4*, CRC Press, 2016.

[8] J. Haas, *A history of the unity game engine,* Diss. Worcester Polytechnic Institute, 2014.

Authors' statement

This article was produced under the project Project RoUa 2SOFT/1.2/86 Ro-Ua Cross-border Academic Development for Research and Innovation”.