ViVo: Visibility-Aware Mobile Volumetric Video Streaming

Bo Han、Yu Liu、Feng Qian

摘要

In this paper, we perform a first comprehensive study of mobile volumetric video streaming. Volumetric videos are truly 3D, allowing six degrees of freedom (6DoF) movement for their viewers during playback. Such flexibility enables numerous applications in entertainment, healthcare, education, etc. However, volumetric video streaming is extremely bandwidth-intensive. We conduct a detailed investigation of each of the following aspects for point cloud streaming (a popular volumetric data format): encoding, decoding, segmentation, viewport movement patterns, and viewport prediction. Motivated by the observations from the above study, we propose ViVo, which is to the best of our knowledge the first practical mobile volumetric video streaming system with three visibility-aware optimizations. ViVo judiciously determines the video content to fetch based on how, what and where a viewer perceives for reducing bandwidth consumption of volumetric video streaming. Our evaluations over real wireless networks (including commercial 5G), mobile devices and users indicate that ViVo can save on average 40% of data usage (up to 80%) with virtually no drop in visual quality.

在这一篇论文中，我们对移动体积视频流进行了首次全面的研究。体积视频是真正的 3D，允许观众在播放期间进行六个自由度（6DoF）的移动。这种灵活性使娱乐、医疗保健、教育等领域的众多应用成为可能。然而，体积视频流非常地占用带宽。我们对点云流（一种受欢迎的体积视频数据格式）的以下每个方面进行了详细调查：编码、解码、分割、视口移动模式和视口预测。在上述研究的观察结果的推动下，我们提出了 ViVo，这是我们已知地第一个具有三种可见性感知优化的实用移动体积视频流系统。ViVo 根据观众感知的方式、内容和位置明智地确定要获取的视频内容，以减少容量视频流的带宽消耗。我们在实际的无线网（包括 5G），移动设备和用户上部署 VIVO 进行评估。VIVO 可以实现降低平均 40% 的数据使用率（最高 80%）的同时保证视觉质量的保真。

关键词：点云视频；分割；预测

1 引言

随着类似 5G 的无线网技术的逐渐成熟，许多新兴的应用也开始不断的出现。其中，沉浸式视频流是一个十分重要的应用。这篇文章中研究了一种新的视频内容形式，叫做体积视频。体积视频是混合现实的一种可行方案，也会成为 5G 时代的一个重要的应用。

与传统视频是由像素组成所不同的是，体积视频是 3D 的。体积视频的每一帧都是由 3D 的点或者网格组成的。在播放期间，观众可以在 6 个自由度上进行自由的观看（3 个位置移动的自由度和 3 个视角旋转的自由度）。体积视频可以应用到许多常规视频不能实现的地方，例如制作外科手术的教学视频，让医学生可以从各个不同的方向观看视频的内容。除了内容形式的不同，体积视频还有许多的地方和常规视频是不一样的。例如制作，传输，编码和分析。目前关于点云视频流的标准化工作还在研究中。

点云视频带来内容和体验的极大提升，伴随而来的是占用存储空间庞大，传输时对带宽的需求也十分的苛刻。于是优化点云视频的传输是一个十分重要的课题。目前对于传输优化的方案主要有分块和服务端渲染。

分块主要是思想是将点云分成一个个小块，这些小块可以独立编码、传输和渲染。观看点云视频时，大多时候是不会看到全部的点云内容的，或者说看到全部的小块。所以传输时只需要将观众即将看到的块传输给客户端即可，从而实现带块的节省。服务端渲染是在服务器端对观众即将看到的画面渲染完成，再将画面发送给客户端进行播放。

这篇论文采用的是分块的思想，通过三个可见性的优化算法选出，得到一个优秀的选择块的方案，实现既满足用户观看体验，又实现带宽的节省。论文的复现工作也围绕着三个优化算法展开。

2 相关工作

点云视频捕获。现在常用的点云视频捕获手段有 RBG-D 摄像机，例如：Microsoft Kinect[1],Intel RealSense[2]，和各种各样的 LIDAR 扫描[3]。这些设备都配备有深度传感器，能够从不同的视角捕获 3D 信息，最后整合成一个 3D 的点云文件。本次的复现工作采用的是开源的数据集进行实验[4]。

数据格式。3D Mesh[5] 和点云是两种较为受欢迎的体积视频呈现方案。3D Mesh 是用一系列的顶点、边和表面来重建 3D 模型的结构。而点云就仅仅是一系列的 3D 的点，这些点具备颜色等属性信息。相比较与 3D Mesh，点云更加的灵活，更加的简单，因为它仅仅包含无组织的点，不需要维护拓扑一致性的信息，可以进行自由的分块操作。

压缩。点云压缩现有的许多工作大多基于八叉树[6]和 k-d 树[7]。有损压缩一般都是按照某种策略删除部分的点，以牺牲细节来实现点云的压缩。在这篇论文的研究中，作者认为人眼对于细节的分辨率是有限的，当两个画面的相似性足够接近时，细节的微小差异不影响用户观看。所以提出对于不同景深的点云块，采用不同的压缩率，在保证用户观看画面质量的前提，尽量节省带宽。

3 本文方法

3.1 点云分块

点云庞大的数据都是因为其点的数量众多。而在传输时，最直接节省带宽的方法就是减少传输的点的数量。而分块是最简单的减少点云的方法。分块是将点云空间中的点按照空间的区域划分成一个个独立的块。每一个块能够单独的传输和渲染。

分块的大小是影响性能的一个因素。当一个块只有一小部分出现在用户视野内，就需要将整个块都传输给播放端，其中包含许多不需要的点。而块越小，对于不需要的点的传输就越少。从这一个角度看，分块越小，节省的带宽越多。但分块越多，在进行是否在视野内部的判断时，需要的开销也就越大，甚至会抵消掉分块带来的收益。因此，分块的大小是一个需要仔细权衡的因素。

文章对 $25*25*25cm^3$、$50*50*50cm^3$ 和 $100*100*100cm^3$ 的分块方法进行了测试。测试结果显示，分块越小，开销越大，并认为原因是分块越小，压缩率越小，最后对带宽的需求越大。作者最后选择的是 $100*100*100cm^3$ 。

3.2 视角预测

作者雇佣了 32 位来自于大学和大公司的用户，从它们哪里采集视角的轨迹数据。这些用户使用的设备有手机和头戴式设备。从这些轨迹数据中，有如下发现：一、几乎所有的运动都在同一个平面上，即用户在垂直方向上的运动很少，可能是因为跳跃和下蹲对于用户来说并不是方便。二、手机用户的移动倾向于直线，因为 6 个自由度的操作并不是很方便；而头戴式设备的用户运动上会更加自然。

相比较于 360° 视频来说，体积视频的视角预测更加的复杂，因为有 6 的自由度需要预测。如果同时考虑 6 个自由度的预测，会因为预测空间巨大而难以实现或难以部署。所以采用对每一个坐标进行单独的预测，然后再整合成一个结果。

为了能够更快的得到预测的结果以及节省计算资源，作者采用的预测方法是现有的线性回归和多层感知算法。假设当前的播放时刻为 T。两个算法都采用 hms 的历史窗口，该窗口包含的 $T-h$ 到 T 时间段帧的视区信息。用这个窗口的中历史信息来训练模型预测 $T+p$ 时刻的视区信息，p 是预测窗口。

作者设置预测窗口为 50ms 到 500ms, 历史窗口设置为 33ms 到 660ms。最终得出结论：当 h 大约是 $p/2$ 时，线性回归算法的准确度最高；而当 h 为 660 ms 时，MLP 的位置预测最准确，66ms 时 MLP 的角度预测最准确。最终，作者因线性回归的轻量级特性而放弃 MLP 算法。

3.3 可见性优化

当获得一个视角的 6 个自由度信息后，就可以得到一个由所谓的视锥体[8], 如图 1。视锥体是摄像机可见的空间，看上去像截掉顶部的金字塔。只有在视锥体内部的物体才会投影到屏幕上。

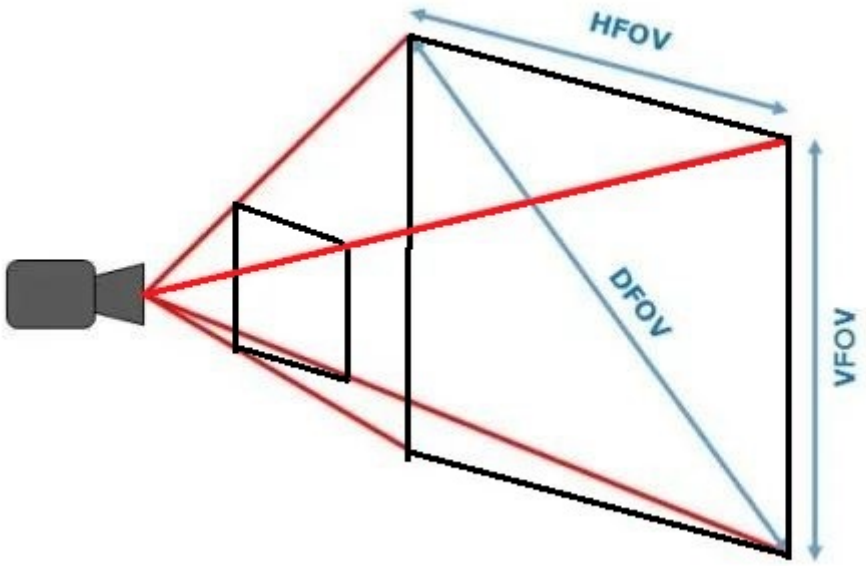


图 1: 视锥体

由于视角的六个自由度信息是分开进行预测的，在整合成一个视角时，存在较大的误差。为了减少误差对用户观看画面的影响，将视角 (FoV) 进行扩大，增加需要抓取的块的范围，增加容错率。文章是通过设置 FoV 的角度为 60°、90° 和 120° 得到三个视锥体。按照这个 FoV 的顺序，对其判定在内部的块进行优先级设置。优先级高的传输无损的点云数据，优先级低的传输相应的压缩过的数据。

3.4 遮挡优化

计算机图形学有相关的点云遮挡问题的算法，如：Hidden Point Removal (HPR) operator[9]。但是些算法大多基于点的处理，计算开销巨大，并且速度缓慢，并不适用于体积视频流问题。所以作者提出了基于块的算法。

将一个块 C 的中心与视角位置相连，考察周围一圈块的遮挡属性。作者认为能够遮挡 c 的邻居具备这样的属性：1. 距离视角位置更近。2. 和这条线有相交。计算出满足遮挡属性的邻居数量 $S(c)$，和点数量最多的邻居 B 与目标块点数比 $R(c) = B/C$。再按照下面的公式计算被遮挡的可能性 $O(c)$。

$$O(c) = \begin{cases} 0 & R(c) < \alpha_0 \beta^{S(c)-1} \\ 1 & \alpha_0 \beta^{S(c)-1} < R(c) < \alpha_1 \beta^{S(c)-1} \\ 2 & \alpha_1 \beta^{S(c)-1} < R(c) < \alpha_2 \beta^{S(c)-1} \\ 3 & \alpha_2 \beta^{S(c)-1 < R(c)} \end{cases} \tag{1}$$

遮挡可能性 $O(c)$ 越大，块的优先级设置越小。作者通过实验选择的最佳参数 $\alpha_0, \alpha_1, \alpha_2, \beta$ 的值分别为 $0.6, 1.0, 3.0, 0.8$

3.5 深度优化

当点云块距离视角越远，则在投影后得到的细节越少，且存在和相邻块重合的可能。此时如果选择高质量的点云块就会产生一部分的性能损失。所以作者提出根据不同的距离选择不同的压缩点云，同时对与视频的观看质量不会造成明显的损伤或者没有损伤, 如图 2。

图 2: 不同深度采用不同的压缩率

作者根据实验的结果选择的距离和压缩率对应方案：[0, 3.2)→100%, [3.2, 4.2)→80%, [4.2, 5.2)→60%, [5.2,6.2)→40%, and [6.2, ∞)→20%.

4 复现细节

4.1 与已有开源代码对比

该论文没有开源代码。复现时使用 unity 实现，其中相机的控制和 PLY 文件读取代码参考了网络上相关的教程和开源的项目。

4.1.1 点云分割与压缩

点云分割与压缩采用的 PCL 库文件提供的相关方法[10]。分割成块是依据空间区域的划分。只需要将点云中点进行区域的判定就可以。

```
void Split(string filename, string temp, int count)
{
    pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud(new pcl::PointCloud<pcl::
        PointXYZRGB>);
    pcl::io::loadPLYFile<pcl::PointXYZRGB>(filename, *cloud);

    pcl::PointCloud<pcl::PointXYZRGB>::Ptr newcloud(new pcl::PointCloud<
        pcl::PointXYZRGB>);

    // 归一化,目标空间 12*12*12
    for (long i = 0; i < cloud->size(); i++)
    {
        cloud->points[i].x = cloud->points[i].x /1024 *12;
        cloud->points[i].y = cloud->points[i].y /1024 *12 ;
        cloud->points[i].z = cloud->points[i].z /1024 *12 ;
    }
    // 分割
    for (int x = 0; x < 6; x++)
    {
        for (int y = 0; y < 6; y++)
        {
            for (int z = 0; z < 6; z++)
            {
                for (long i = 0; i < cloud->size(); i++)
                {
                    if (x * 2 < cloud->points[i].x && cloud->points[i].x < (x
                        +1)*2 && \
                        y * 2 < cloud->points[i].y && cloud->points[i].y < (y
                            +1)*2 && \
                        z * 2 < cloud->points[i].z && cloud->points[i].z < (z
                            +1)*2 )
                    {

                        newcloud->push_back(cloud->points[i]);
                    }
                }
                if (newcloud->size() == 0)
                {
                    cout << " this tile have no point" << endl;
                }
                else
                {
                    string f = temp + "/" + to_string(count + 1) + "_" +
                        to_string(x + 1) + to_string(y + 1) + to_string(z + 1);
                    cout << f << endl;
```

```
                Compress(newcloud, f, 0);
                for (int i = 0; i < 5; i++)
                {
                    Compress(newcloud, f, i);
                }
            }
            // 维护一个描述tile点的文件
            if (count % 5 == 0)
            {
                fstream file_1;
                fstream file_2;
                file_1.open(point_data, std::ios_base::app | std::ios_base
                    ::in);
                file_2.open(point_data_b, std::ios::binary|std::ios_base::
                    app | std::ios_base::in);
                if (file_1.is_open())
                {
                    file_1 << newcloud->size() << "\t" ;
                }
                if (file_2.is_open())
                {
                    int temp = newcloud->size();
                    file_2.write(reinterpret_cast<char *>(&temp), sizeof(
                        int));
                }
                file_1.close();
                file_2.close();
            }
            newcloud.reset(new pcl::PointCloud<pcl::PointXYZRGB>);
        }
    }

}
}
```

4.1.2 相机控制

在点云播放时，相机是模拟人眼的组件。为模拟人眼观看点云视频，需要对相机的运动进行控制。这部分代码采用的是网上现有的资源[11]。

4.1.3 点云文件读取

整个复现的工作聚焦于文章提出的 3 个优化算法，并没有部署到网络的传输。所以点云视频播放需要读取本地的文件。这部分代码参考 Pcx 的项目[12]。

4.1.4 视角预测

视角预测采用简单的线性回归算法。这部分代码由自己编写完成。

```
    private void predict_1()
    {
        CameraState[] his = history.ToArray();

        // 初始时，历史窗口数据不够
        if (history.Count < Max)
```

```
        {
            pre_cam = his[history.Count - 1]; // 直接返回最新的视角
            return;
        }

        // 初始化直线辅助参数，一共6条，a,b 公用参数，c,d 各自有
        float a = 0;
        float b = 0;
        float[] c = new float[] { 0, 0, 0, 0, 0, 0 };
        float[] d = new float[] { 0, 0, 0, 0, 0, 0 };
        // 计算辅助参数
        for (int i = 0; i < Max; i++)
        {
            a += i * i;
            b += i;
        }
        for (int i = 0; i < Max; i++)
        {
            c[0] += i * his[i].x;
            c[1] += i * his[i].y;
            c[2] += i * his[i].z;
            c[3] += i * his[i].yaw;
            c[4] += i * his[i].pitch;
            c[5] += i * his[i].roll;
            d[0] += his[i].x;
            d[1] += his[i].y;
            d[2] += his[i].z;
            d[3] += his[i].yaw;
            d[4] += his[i].pitch;
            d[5] += his[i].roll;

        }

        // 计算直线
        float[] k = new float[6];
        float[] t = new float[6];
        for (int i = 0; i < 6; i++)
        {
            k[i] = (c[i] * 3 - b * d[i]) / (a * 3 - b * b);
            t[i] = (a * d[i] - c[i] * b) / (a * 3 - b * b);
        }

        // 计算预测值
        pre_cam.x = (Max + prewin) * k[0] + t[0];
        pre_cam.y = (Max + prewin) * k[1] + t[1];
        pre_cam.z = (Max + prewin) * k[2] + t[2];
        pre_cam.yaw = (Max + prewin) * k[3] + t[3];
        pre_cam.pitch = (Max + prewin) * k[4] + t[4];
        pre_cam.roll = (Max + prewin) * k[5] + t[5];
    }
```

4.1.5 三个优化算法

三个优化算法是参考文章提出的算法思路编写完成。

```
public void PV(GameObject cam)
    {
        pv_result.Clear();
        for (int i = 1; i < 4; i++)
```

```
        {
            cam.GetComponent<Camera>().fieldOfView = 30 * i;
            GeometryUtility.CalculateFrustumPlanes(cam.GetComponent<Camera
                >(), planes);
            for (var index = 0; index < obj_list.Count; index++)
            {
                var bounds = obj_list[index].GetComponent<Renderer>().bounds;
                var result = GeometryUtility.TestPlanesAABB(planes, bounds);
                    // 判断是否在视锥内
                Tile_level temp = new Tile_level(bounds, initial_level - i);
                if (result && !pv_result.Exists(t => t.bound == temp.bound))
                {
                    pv_result.Add(temp);
                }
            }
        }
    }
```

```
public void OV(GameObject cam)
{
        ov_result.Clear();

        string path = "F:/PLY/point_num_b.txt"; //存放点数量的文件
        int tile_len = (int)(Math.Pow(tile_sum, 3));
        int[] point_number = new int[tile_len];
        BinaryReader br;
        br = new BinaryReader(new FileStream(path, FileMode.Open));
        for (int i = 0; i < point_number.Length; i++)
        {
            point_number[i] = br.ReadInt32();
        }
        br.Close();
        // 算法中的 α 0, 1, 2 和 β
        double[] parament = new double[4] { 0.6, 1, 3, 0.8};

        for (var index = 0; index < pv_result.Count; index++)
        {
            var bound_temp = pv_result[index].bound;
            // 获取块的点的数量
            Vector3 tile_position = (bound_temp.center - new Vector3(
                tile_size / 2, tile_size / 2, tile_size / 2)) / tile_size;
            int tile_point = point_number[((int)(tile_position.x * tile_sum
                * tile_sum + tile_position.y * tile_sum + tile_position.z))];
            if(tile_point == 0)
            {
                continue;
            }
            RaycastHit[] hits;
            hits = Physics.RaycastAll(bound_temp.center, cam.transform.
                position - bound_temp.center, tile_size);
            int s = hits.Length;                //算法中的 S(c)
            int max_point = 0;

            for(int i =0;i<s; i++)
            {
                Vector3 temp_ov = (hits[i].transform.position - new Vector3(
                    tile_size / 2, tile_size / 2, tile_size / 2)) / tile_size;
                int k = point_number[((int)(temp_ov.x * tile_sum * tile_sum +
                     temp_ov.y * tile_sum + temp_ov.z))];
                if (k > max_point)
```

```
            {
                max_point = k;
            }
        }
        float r = max_point / tile_point; // 算法中的 R(c)
        // 设置遮挡可能性的优先级
        double temp = Math.Pow(parament[3], s - 1);
        int d_level = 0;
        if (r < parament[0] * temp)
        {
            d_level = 0;
        }
        else if(r< parament[1] * temp)
        {
            d_level = 1;
        }
        else if(r< parament[2] * temp)
        {
            d_level = 2;
        }
        else
        {
            d_level = 3;
        }
        Tile_level ov_res = new Tile_level(pv_result[index].bound,
            pv_result[index].level - d_level);
        ov_result.Add(ov_res);
    }
}
```

```
public void DV(GameObject cam)
{
    dv_result.Clear();
    ov_result = ov_result.OrderByDescending(t => t.level).ToList(); // 按照优先级降序排列。
    float[] distance_threshold =new float[]{ 19.2f, 25.2f, 31.2f, 37.2f }; // 设置5个距离的阈值，注意从实际空间到unity空间的映射
    for (var index = 0; index < ov_result.Count; index++)
    {
        if(ov_result[index].level <= 0)
        {
            break;
        }
        var bound_temp = ov_result[index].bound;
        float dis1 = (cam.transform.position - bound_temp.center).magnitude;

        if(dis1 < distance_threshold[0])
        {
            dv_result.Add(new Tile_level(ov_result[index].bound, 5));
        }
        else if(dis1 < distance_threshold[1])
        {
            dv_result.Add(new Tile_level(ov_result[index].bound, 4));
        }
        else if(dis1 < distance_threshold[2])
        {
            dv_result.Add(new Tile_level(ov_result[index].bound, 3));
        }
        else if(dis1 <distance_threshold[3])
```

```
            {
                dv_result.Add(new Tile_level(ov_result[index].bound, 2));
            }
            else
            {
                dv_result.Add(new Tile_level(ov_result[index].bound, 1));
            }
        }
    }
```

根据三个优化结果获取相应的 tile 文件列表。

```
public void Get_tile_number(List<string> tile_number)
    {
        tile_number.Clear();
        for (var index = 0; index < dv_result.Count; index++)
        {
            if (dv_result[index].level > 0)
            {
                int level_temp = dv_result[index].level * 20;
                var bound_temp = dv_result[index].bound;
                Vector3 tile_position = (bound_temp.center + new Vector3(
                    tile_size / 2, tile_size / 2, tile_size / 2)) / tile_size;
                string tile_name = tile_position.x.ToString("0") +
                                   tile_position.y.ToString("0") +
                                   tile_position.z.ToString("0") +
                                   "_" + level_temp.ToString() + ".ply";
                tile_number.Add(tile_name);
            }
        }
    }
```

4.2 实验环境搭建

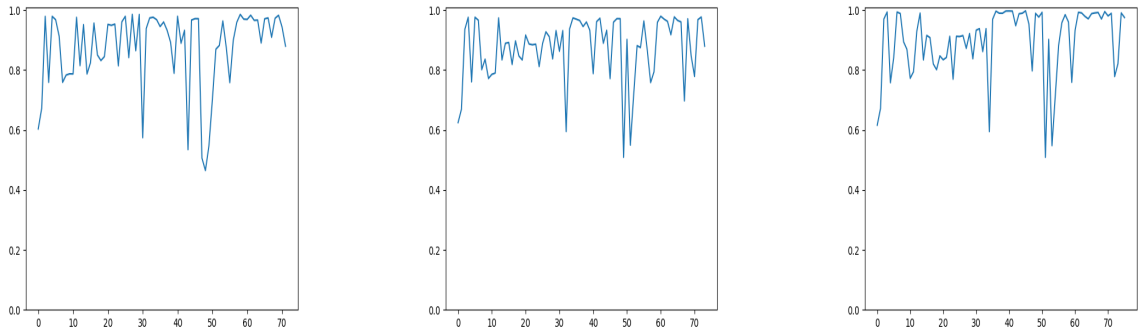
复现工作可以粗略的分为两个部分。一个是准备阶段，一个是复现阶段。准备阶段是使用 PCL 库对原始的点云数据进行处理，如分块和压缩。复现阶段就是对文章提出的优化算法进行实现，然后设计相应的实验验证复现结果的性能和文章的性能差异。

准备阶段是调用 PCL 库实现对点云数据的处理，用 C++ 开发，相关的环境搭建可以参考网上的资源。复现阶段在 Unity 环境中进行，使用 C# 开发相应脚本。

5 实验结果分析

实验分可行性验证和带宽节省性能验证。可行性验证是截取加载全部的点云时生成的画面和算法优化后截取的画面，计算两个画面的结构相似性 (SSIM)。SSIM 是验证图片的相似性指标，其值越高，两个图片的相似性就越大，SSIM = 1 时，图片是完全相同的。

5.1 可行性验证



(a) 只进行一个预测可见性优化　(b) 预测和遮挡可见性优化　(c) 同时使用三个优化

图 3: 实验结果

图 3展示的是可行性实验的结果。该实验是预先记录一个相机轨迹，然后用这条轨迹来分析各个算法的可行性。从图中的结果可以看到大多数的时候 SSIM 都能保持到 0.8 以上，但也有部分的时候 SSIM 出现低于 0.7，这是因为预测的准确度不够，出现偏差。

5.2 性能验证

由于没有部署到网络上，在计算带宽需求时是通过对比原始视频数据和播放所需块数据的大小。

实验	原始数据大小	块数据大小
预测可见性	2.1G	890M
预测和遮挡	2.2G	909M
使用三个优化	2.1G	904M

表 1: 播放时需要的文件大小

从表 1中可以看到，基本可以实现带宽节省一半的带宽。论文中手机实验是节省 40% 左右，头戴式设备可以达到 70%。因为头戴式设备的运动更加平滑，预测的效果更好。另外，复现工作效果更好是因为预测采用的相机 FoV 是 30°，60° 和 90°，可见性范围判定更加小。论文中提及，增加遮挡和深度的优化能再提升 5% 左右性能。与预测可见性优化相比增益较少。而这在复现工作中并没有十分明显的体现。个人认为是因为数据集和相机轨迹不够丰富，实验缺乏多样性。

6 总结与展望

通过对这篇论文的复现，使得我对于点云视频的理解更进一步。复现工作中出现的问题也会成为我进步的经验。这里简单总结以下收获。

虽然并没有完整的复现文章中的整个工作，但是却切切实实的感受到一个研究的过程，让我对研究有一个更具体地了解。同时，在老师的指导下设计相关实验，以及学习如何评价复现工作是否正确、有没有没有脱离文章的思路。并且学会在复现过程中思考作者方法的创新型和局限性。

另外，本科非计算机专业毕业的我在这一次也学到了许多关于编程的小技巧，这里十分感谢我的师兄师姐提供的帮助，为我提供技术上的指引。

最后，复现工作中的三个优化算法初步实现，但是研究的工作并不会因此而停止。如，复现过程中出现的分块大小问题没有更近一步的探索；对遮挡问题关于非相邻块的判断没有研究；对于部署到

网络上进行更进一步的研究也没有实现。这些问题都是以后需要研究需要注意和完善的。复现的过程中也注意到对于点云视频的研究现状的了解还是比较匮乏，之后一段时间会结合复现中出现的问题，阅读相应的文章，增加对点云视频研究现状的了解。

参考文献

[1] Kinect for Windows[EB/OL]. https://developer.microsoft.com/en-us/windows/kinect.

[2] Intel RealSense Technology[EB/OL]. https://www.intel.com/content/www/us/en/architecture-and-technology/realsense-overview.html.

[3] QIU H, AHMAD F, BAI F, et al. AVR: Augmented Vehicular Reality[C]//. Association for Computing Machinery, 2018.

[4] 8i introduces fully volumetric 3D video[EB/OL]. https://www.youtube.com/watch?v=aO3TAke7_MI.

[5] MAGLO A, LAVOUÉ G, DUPONT F, et al. 3D Mesh Compression: Survey, Comparisons, and Emerging Trends[J]. ACM Comput. Surv., 2015.

[6] GOLLA T, KLEIN R. Real-time point cloud compression[J]. 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2015: 5087-5092.

[7] LIEN J M, KURILLO G, BAJCSY R. Multi-camera tele-immersion system with real-time model driven data compression[J]. The Visual Computer, 2009, 26: 3-15.

[8] View Frustum Culling[EB/OL]. www.lighthouse3d.com/tutorials/view-frustum-culling/.

[9] KATZ S, TAL A, BASRI R. Direct visibility of point sets[J]. ACM SIGGRAPH 2007 papers, 2007.

[10] Point Cloud Library[EB/OL]. https://pointclouds.org/.

[11] Unity 相机自由移动[EB/OL]. https://blog.csdn.net/weixin_44350205/article/details/99675552.

[12] Pcx - Point Cloud Importer/Renderer for Unity[EB/OL]. https://github.com/keijiro/Pcx.