Run-Length Compression of Large Sparse Potential Visible Sets

J.M.P. van Waveren

May 22nd 2007

Abstract

An efficient algorithm is presented for lossless compression of very large Potentially Visible Sets with high occlusion ratios.

1. Potential Visible Set

A Potentially Visible Set (PVS) is a directional mapping that defines visibility between pairs of objects and/or locations. The mapping is directional because if B is visible to A then A is not necessarily visible to B. Potential Visible Sets are often based on convex hulls or areas that are created with a Binary Space Partitioning (BSP) algorithm. In the context of PVS data, these convex hulls or areas are typically referred to as cells.

A PVS is particularly useful to speed up rendering solutions because it allows many cells or objects that can never be seen from a particular location to be quickly discarded [1]. Although a PVS is typically associated with rendering, it can also be used for other purposes. A PVS can for instance improve the performance of a networking solution [4] by using the PVS to determine which objects need to be synchronized over a network based on visibility.

A PVS can also be used in a context where the term visibility does not necessarily refer to sight. A PVS can for instance be used to quickly cull objects for an obstacle avoidance solution for robots or artificial intelligence (AI). A building may be subdivided into rooms filled with dynamic obstacles, and for obstacle avoidance purposes it may be necessary to quickly gather all objects that need to be considered as obstacles in the current room and nearby rooms. In this context, nearby rooms are not necessarily visible, and also not necessarily nearby based on Euclidean distance. For instance, a path around obstacles may lead through rooms that are not visible and rooms below or above the robot or AI may be physically nearby but may not need to be considered for obstacle avoidance because they are out of reach. In the context of obstacle avoidance "visibility" typically refers to being reachable within a certain distance or time.

PVS data is typically stored as a set of bit strings. In such a bit string the offset of a bit denotes a cell index. A bit set to one denotes visible, and a bit set to zero denotes not visible. To store directional visibility information for all pairs of cells, these bit strings are either the rows or columns in a visibility matrix. The rows typically store all cells visible to a cell, and the columns

store all the cells from which a cell is visible. Even though only one bit is stored for each directed pair of cells the matrix may still consume a large amount of memory if there are thousands of cells.

Clusters of cells can be used to shrink the PVS data [2, 3]. Instead of storing the visibility of cells the visibility of groups of cells is stored. Because there are fewer clusters than cells, the amount of PVS data is reduced. Using clusters can be considered a lossy form of compression because some visibility information between individual cells may be lost. Using clusters may also come at a cost. Relevant PVS dependent data is often stored per cell and a mapping from cells to clusters and back may be required to access this data. Such a mapping consumes memory by itself.

The cells can also be reordered such that longer sequences of zero bits are created that typically compress better. However, finding the optimal order of cells is a complex optimization problem. The following sections focus on algorithms for lossless compression of PVS bit strings without reordering the cells.

2. Bit String Compression

The decompression of compressed PVS data needs to be fast and simple because a PVS is typically used to speed up or avoid more expensive computations. In particular iterating over all cells visible to a cell needs to be quick and easy.

Applying an entropy encoder like Huffman is not optimal because the PVS data is not a sequence of fixed bit size words. For the same reason arithmetic coding of a PVS bit string does not work well. When a PVS bit string is chopped up in fixed size words, the distribution is very uneven where there are many words with only zero bits and all non-zero words may appear at similar frequencies.

PVS data consists of many zeros and long sequences of zeros but there may also be short runs of bits with seemingly random values. The best PVS compression ratios are typically achieved by using Run-Length Encoding (RLE). If the most frequent symbol is precisely known, RLE can be improved to encode only sequences of this symbol and leave all others uncoded. In the case of PVS data the obvious symbol is a single zero bit or some sequence of zero bits like a zero byte.

3. Quake PVS Compression

The computer game Quake used a very simple byte based RLE algorithm to compress the PVS bit strings used for both rendering and networking. This algorithm writes out a byte in whole to the compressed stream if a byte is unequal zero. If a byte is zero then first a zero byte is written to the compressed stream followed by a byte that represents the number of consecutive bytes also set to zero.

```
typedef unsigned char byte;

int CompressPVS( const byte *pvs, int numBytes, byte *compressed ) {
    byte * out = compressed;
    for ( int i = 0; i < numBytes; i++ ) {
        *out++ = pvs[i];
        if ( pvs[i] ) {
            continue;
        }
        int c = 0;
        for ( i++; i < numBytes; i++ ) {
            if ( pvs[i] || c == 256 ) {
                break;
            }
            c++;
        }
        *out++ = c;
        i--;
    }
    return out - compressed;
}

void DecompressPVS( byte *pvs, int numBytes, const byte *compressed ) {
    const byte * in = compressed;
    byte * out = pvs;
    do {
        if ( in[0] ) {
            *out++ = *in++;
            continue;
        }
        for ( int c = in[1] + 1; c != 0; c-- ) {
            *out++ = 0;
        }
        in += 2;
    } while( out - pvs < numBytes );
}
```

The compressed data is byte aligned which makes decompression simple and fast. Worst case the compressed data is 1.5 times larger than then the uncompressed data when the PVS bit string is a sequence of bytes with values that alternate between zero and not zero. The encoder works really well for shorter bit strings (< 5000 cells) with occlusion percentage of 80% to 95%. For longer bit strings with sequences of more than 2048 consecutive zero bits (= 256 * 8-bits) the compression can be improved.

4. Compression of Very Long PVS Bit Strings

The following algorithm is the result of a first attempt at trying to implement an encoder that more efficiently deals with very long sequences of zero bits. The bit string is encoded using blocks of 16 bits, where the first 12 bits represent the number of consecutive zero bits and the next 4 bits represent the number of consecutive one bits that follow the sequence of zero bits. By using 12 bits to encode sequences of zero bits the encoder is more efficient when there are very long sequences of more than 2048 zero bits. In particular the encoder can encode sequences of up to 2^12 = 4096 zero bits more efficiently.

```
#define PVS_RLE_ZERO_BITS       12
#define PVS_RLE_ONE_BITS        4

int CompressPVS( const byte *pvs, int numBytes, byte *compressed ) {
    byte *out = compressed;
    int numBits = numBytes * 8;
    for ( int index = 0; index < numBits; ) {
        int numZeros;
        for ( numZeros = 0; index + numZeros < numBits && numZeros < ( ( 1 << PVS_RLE_ZERO_BITS ) - 1 ); numZeros++ ) {
            if ( ( pvs[( index + numZeros ) >> 3] & ( 1 << ( ( index + numZeros ) & 7 ) ) ) != 0 ) {
                break;
            }
        }
        index += numZeros;
        int numOnes;
        for ( numOnes = 0; index + numOnes < numBits && numOnes < ( ( 1 << PVS_RLE_ONE_BITS ) - 1 ); numOnes++ ) {
            if ( ( pvs[( index + numOnes ) >> 3] & ( 1 << ( ( index + numOnes ) & 7 ) ) ) == 0 ) {
                break;
            }
        }
        index += numOnes;
        int rleBits = numZeros | ( numOnes << PVS_RLE_ZERO_BITS );
        *out++ = rleBits & 255;
        *out++ = rleBits >> 8;
    }
    return out - compressed;
}

void DecompressPVS( byte *pvs, int numBytes, const byte *compressed ) {
    memset( pvs, 0, numBytes );
    int numBits = numBytes * 8;
    for ( int offset = 0, index = 0; index < numBits; ) {
        int rleBits = compressed[offset++] | ( compressed[offset++] << 8 );
        int numZeros = rleBits & ( ( 1 << PVS RLE ZERO BITS ) - 1 );
        index += numZeros;
        int numOnes = rleBits >> PVS_RLE_ZERO_BITS;
        for ( int i = 0; i < numOnes; i++ ) {
            pvs[index >> 3] |= 1 << ( index & 7 );
            index++;
        }
    }
}
```

This encoder sometimes works better than the Quake encoder but quite often also worse than the Quake encoder. In particular the compression does not work well if there are sequences of alternating bits. The encoder works better if there are very long sequences of consecutive zero bits of more than 2048 bits. This compressor shows that it is important to be able to efficiently encode bit sequences of consecutive zeros that are longer than 2048 bits (otherwise this compressor would never outperform the Quake compressor). However, it is also important to be able to store immediate values with alternating bit sequences directly from the PVS bit string.

The compressor shown below still produces byte aligned data which allows for simple and fast decompression. However, the following compressor not only allows longer sequences of zero bits to be encoded but also allows immediate values to be placed directly into the compressed data.

The following algorithm compresses PVS bit strings such that if the first bit of a byte of compressed data is set to zero the next 7 bits contain an immediate value with actual PVS bits. If the first bit is set to one the next 7 bits and possibly the next byte of the compressed data store a run of zeros. If the first bit is set to one and the second bit is set to zero then the next 6 bits store the number of consecutive bits set to zero. If the first bit and the second bit are both set to one then the next 6 bits plus the next 8 bits of compressed data store the number of consecutive bits set to zero.

```
#define PVS_RLE_IMMEDIATE_BITS     7   // number of bits available to encode an immediate
#define PVS_RLE_1ST_COUNT_BITS     6   // default number of bits to encode a run of zeros
#define PVS_RLE_2ND_COUNT_BITS     8   // one additional byte to encode a run of zeros
#define PVS_RLE_RUN_GRANULARITY    1   // can be set to a higher value if runs of more
                                       // than 16384 zeros are common
#define PVS_RLE_RUN_BIT            (1 << 7)
#define PVS_RLE_RUN_LONG_BIT       (1 << 6)

int CompressPVS( const byte *pvs, int numBytes, byte *compressed ) {
    byte *out = compressed;
    int numBits = numBytes * 8;
    for ( int index = 0; index < numBits; ) {
        int numNotVis;
        for ( numNotVis = 0; index + numNotVis < numBits; numNotVis++ ) {
            if ( ( pvs[( index + numNotVis ) >> 3] &
                                ( 1 << ( ( index + numNotVis ) & 7 ) ) ) != 0 ) {
                break;
            }
        }
        if ( numNotVis >= PVS RLE IMMEDIATE BITS ) {
            if ( numNotVis > ( 1 << PVS_RLE_1ST_COUNT_BITS ) * PVS_RLE_RUN_GRANULARITY ) {
                // run of zeros of ( 1 << ( PVS RLE 1ST COUNT BITS + PVS RLE 2ND COUNT BITS ) ) *
                // PVS_RLE_RUN_GRANULARITY bits
                if ( numNotVis > ( 1 << ( PVS_RLE_1ST_COUNT_BITS + PVS_RLE_2ND_COUNT_BITS ) ) *
                                    PVS_RLE_RUN_GRANULARITY ) {
                    numNotVis = ( 1 << ( PVS RLE 1ST COUNT BITS + PVS RLE 2ND COUNT BITS ) );
                } else {
                    numNotVis /= PVS RLE RUN GRANULARITY;
                }
                *out++ = ( ( ( numNotVis - 1 ) & ( ( 1 << PVS_RLE_1ST_COUNT_BITS ) - 1 ) ) |
                                ( PVS_RLE_RUN_BIT | PVS_RLE_RUN_LONG_BIT ) );
                *out++ = ( ( ( numNotVis - 1 ) >> PVS RLE 1ST COUNT BITS ) );
            } else {
                // run of zeros of ( 1 << PVS RLE 1ST COUNT BITS ) * PVS RLE RUN GRANULARITY bits
                numNotVis /= PVS_RLE_RUN_GRANULARITY;
                *out++ = ( ( numNotVis - 1 ) | PVS_RLE_RUN_BIT );
            }
            index += numNotVis * PVS_RLE_RUN_GRANULARITY;
        } else {
            // immediate of PVS_RLE_IMMEDIATE_BITS bits
            int bits = 0;
            for ( int j = 0; j < PVS_RLE_IMMEDIATE_BITS && index + j < numBits; j++ ) {
                if ( ( pvs[( index + j ) >> 3] & ( 1 << ( ( index + j ) & 7 ) ) ) != 0 ) {
                    bits |= 1 << j;
                }
            }
            *out++ = bits;
            index += PVS_RLE_IMMEDIATE_BITS;
        }
    }
    return out - compressed;
}
```

```
void DecompressPVS( byte *pvs, int numBytes, const byte *compressed ) {
    memset( pvs, 0, numBytes );
    int numBits = numBytes * 8;
    for ( int offset = 0, index = 0; index < numBits; ) {
        int rleBits = compressed[offset++];

        if ( ( rleBits & PVS_RLE_RUN_BIT ) != 0 ) {

            // short run-length code
            int run = rleBits & ( ( 1 << PVS_RLE_1ST_COUNT_BITS ) - 1 );

            if ( ( rleBits & PVS_RLE_RUN_LONG_BIT ) != 0 ) {
                // additional bits for long run-length code
                run |= compressed[offset++] << PVS RLE 1ST COUNT BITS;
            }

            index += ( run + 1 ) * PVS_RLE_RUN_GRANULARITY;

        } else {

            for ( int i = 0; i < PVS_RLE_IMMEDIATE_BITS; i++ ) {
                pvs[index >> 3] |= ( ( rleBits >> i ) & 1 ) << ( index & 7 );
                index++;
            }
        }
    }
}
```

This compressor can efficiently encode sequences of up to 2^(6+8) = 16384 zero bits while still being able to store immediate values with arbitrary bit sequences directly in the compressed data. The compressed PVS data for 10000 or more cells with an occlusion ratio of 98% or more is some 20% to 30% smaller than the compressed data produced by the run-length encoder used in Quake. Worst case the compressed data can be at most 1.125 times larger than the uncompressed data when the PVS bit string contains no runs of 8 or more consecutive zero bits. The encoder is more complex but encoding is typically done off-line. The decoder is still surprisingly simple and typically not noticeably slower than the run-length decoder from Quake.

5. Results

The following table shows the performance of the different PVS compression algorithms for the PVS data used for rendering and networking in the computer game Quake. The number of cells is relatively small ranging from a couple of hundred to just over 1400. There are two cases where the Quake PVS compression algorithm outperforms the algorithm introduced here. However, in both cases the number of cells is very low and the difference in compressed size is less than 250 bytes. In all other cases the PVS compression algorithm introduced here performs marginally better providing up to a 13% improvement over the PVS compression algorithm used in Quake.

name	# cells	occlusion ratio	uncompressed (bytes)	Quake (bytes / ratio)	New (bytes / ratio)	percentage smaller
dm1.bsp	449	79.8%	25,593	12,595 / 2:1	11,561 / 2:1	8%
dm2.bsp	934	91.1%	109,278	27,559 / 3:1	24,861 / 4:1	10%
dm3.bsp	598	87.5%	44,850	15,741 / 2:1	14,424 / 3:1	8%
dm4.bsp	399	80.1%	19,950	10,863 / 1:1	9,490 / 2:1	13%
dm5.bsp	478	81.7%	28,680	13,882 / 2:1	12,854 / 2:1	7%
dm6.bsp	601	76.5%	45,676	18,156 / 2:1	18,401 / 2:1	-1%
e1m1.bsp	1148	89.8%	165,312	40,843 / 4:1	39,380 / 4:1	4%
e1m2.bsp	1109	91.1%	154,151	31,713 / 4:1	30,294 / 5:1	4%
e1m3.bsp	967	90.9%	117,007	25,516 / 4:1	24,219 / 4:1	5%
e1m4.bsp	1230	91.4%	189,420	37,621 / 5:1	36,130 / 5:1	4%
e1m5.bsp	932	93.2%	109,044	21,204 / 5:1	19,623 / 5:1	7%
e1m6.bsp	648	86.0%	52,488	18,590 / 2:1	17,596 / 2:1	5%
e1m7.bsp	281	67.0%	10,116	6,042 / 1:1	6,266 / 1:1	-4%
e1m8.bsp	508	82.4%	32,512	16,259 / 1:1	14,616 / 2:1	10%
e2m1.bsp	1092	92.7%	149,604	34,723 / 4:1	31,290 / 4:1	10%
e2m2.bsp	1256	92.9%	197,192	44,302 / 4:1	39,502 / 4:1	11%
e2m3.bsp	1095	93.4%	150,015	33,401 / 4:1	29,955 / 5:1	10%
e2m4.bsp	1258	93.4%	198,764	43,076 / 4:1	38,188 / 5:1	11%
e2m5.bsp	985	91.0%	122,140	26,195 / 4:1	25,532 / 4:1	3%
e2m6.bsp	897	93.4%	101,361	22,701 / 4:1	19,968 / 5:1	12%
e2m7.bsp	1236	92.6%	191,580	34,104 / 5:1	32,737 / 5:1	4%
e3m1.bsp	1000	93.5%	125,000	30,685 / 4:1	26,715 / 4:1	13%
e3m2.bsp	522	89.7%	34,452	10,387 / 3:1	9,714 / 3:1	6%
e3m3.bsp	852	90.5%	91,164	19,692 / 4:1	18,982 / 4:1	4%
e3m4.bsp	1166	92.7%	170,236	32,555 / 5:1	31,720 / 5:1	3%
e3m5.bsp	1423	91.9%	253,294	49,973 / 5:1	47,542 / 5:1	5%
e3m6.bsp	1403	92.6%	246,928	44,170 / 5:1	42,659 / 5:1	3%
e3m7.bsp	1021	92.6%	130,688	24,903 / 5:1	23,735 / 5:1	5%
e4m1.bsp	1218	91.3%	186,354	38,730 / 4:1	36,864 / 5:1	5%
e4m2.bsp	1067	91.4%	142,978	30,978 / 4:1	28,373 / 5:1	8%
e4m3.bsp	918	92.1%	105,570	24,335 / 4:1	22,499 / 4:1	8%
e4m4.bsp	1255	91.5%	197,035	41,401 / 4:1	39,124 / 5:1	5%
e4m5.bsp	959	92.0%	115,080	22,745 / 5:1	21,615 / 5:1	5%
e4m6.bsp	804	90.1%	81,204	18,593 / 4:1	17,759 / 4:1	4%
e4m7.bsp	1436	93.6%	258,480	42,378 / 6:1	39,423 / 6:1	7%
e4m8.bsp	988	94.4%	122,512	20,899 / 5:1	19,376 / 6:1	7%

The following table shows the performance of the different PVS compression algorithms for the PVS data used to quickly gather obstacles for dynamic obstacle avoidance in the computer game Enemy Territory Quake Wars (ETQW). The number of cells is significantly higher than the number of cells in the computer game Quake. One of the ETQW environments is subdivided into over 17 thousand cells. The occlusion ratios for the PVS data in ETQW are also significantly

higher than in Quake. All occlusion ratios for the PVS data in ETQW are over 98% and the average occlusion ratio is around 99%. The algorithm introduced here performs noticeably better on these data sets than the PVS compression algorithm from Quake. For one of the ETQW environments the compressed PVS data is 30% smaller when compressed with the algorithm introduced here.

name	# cells	occlusion ratio	uncompressed (bytes)	Quake (bytes / ratio)	New (bytes / ratio)	percentage smaller
area22.aas_player	9390	99.2%	11,023,860	436,506 / 25:1	341,253 / 32:1	22%
ark.aas_player	7339	99.3%	6,737,202	259,172 / 25:1	203,973 / 33:1	21%
canyon.aas_player	10640	99.2%	14,151,200	500,412 / 28:1	384,804 / 36:1	23%
island.aas_player	8779	99.2%	9,639,342	346,037 / 27:1	266,556 / 36:1	23%
outskirts.aas_player	12203	99.4%	18,621,778	626,072 / 29:1	463,395 / 40:1	26%
quarry.aas_player	8878	98.8%	9,854,580	439,844 / 22:1	363,694 / 27:1	17%
refinery.aas_player	9564	99.4%	11,438,544	374,780 / 30:1	281,699 / 40:1	25%
salvage.aas_player	9577	99.0%	11,473,246	493,566 / 23:1	395,918 / 28:1	20%
sewer.aas_player	6089	99.1%	4,639,818	201,705 / 23:1	166,946 / 27:1	17%
slipgate.aas_player	17005	99.5%	36,152,630	948,119 / 38:1	662,510 / 54:1	30%
valley.aas_player	9348	99.2%	10,927,812	414,433 / 26:1	323,519 / 33:1	22%
volcano.aas_player	7641	98.8%	7,304,796	384,153 / 19:1	316,817 / 23:1	18%
area22.aas_vehicle	8352	99.5%	8,719,488	256,040 / 34:1	186,020 / 46:1	27%
ark.aas_vehicle	4506	99.2%	2,541,384	106,599 / 23:1	85,844 / 29:1	19%
canyon.aas_vehicle	8349	99.5%	8,716,356	259,717 / 33:1	188,431 / 46:1	27%
island.aas_vehicle	6598	99.4%	5,443,350	172,659 / 31:1	129,434 / 42:1	25%
outskirts.aas_vehicle	6045	99.3%	4,570,020	165,447 / 27:1	130,017 / 35:1	21%
quarry.aas_vehicle	5419	99.2%	3,674,082	143,090 / 25:1	114,334 / 32:1	20%
refinery.aas_vehicle	5489	99.3%	3,770,943	129,781 / 29:1	100,481 / 37:1	23%
salvage.aas_vehicle	4364	98.7%	2,382,744	126,369 / 18:1	107,168 / 22:1	15%
sewer.aas_vehicle	4432	99.3%	2,455,328	100,904 / 24:1	82,462 / 29:1	18%
slipgate.aas_vehicle	7067	99.3%	6,247,228	205,473 / 30:1	156,567 / 39:1	24%
valley.aas_vehicle	6391	99.4%	5,106,409	171,173 / 29:1	131,517 / 38:1	23%
volcano.aas_vehicle	3928	98.3%	1,928,648	119,733 / 16:1	104,155 / 18:1	13%

6. Conclusion

PVS data for no more than a couple of thousand cells can be efficiently compressed with the lossless compression algorithm used in the computer game Quake. For a couple of hundred to a couple of thousand cells this algorithm performs rather well considering its simplicity. PVS data for many thousands of cells with high occlusion ratios is best compressed with the algorithm introduced here. The encoder is more complex but encoding is typically done off-line while the decoder is still surprisingly simple. For many thousands of cells and high occlusion ratios the PVS compression algorithm introduced here produces 10% to 30% smaller data sets than the PVS compression algorithm used in the computer game Quake.

7. References

1. Inside Quake: Visible-Surface Determination
   Michael Abrash
   Dr. Dobb's Sourcebook, January/February 1996, #255
   Ramblings In Real Time, pp. 41-45
   Available Online: http://www.bluesnews.com/abrash/chap64.shtml
2. Effective Compression Techniques for Precomputed Visibility
   Michiel van de Panne, A. James Stewart
   Eurographics Workshop on Rendering, pp. 305-316, June 1999
   Available Online: http://www.cs.queensu.ca/~jstewart/papers/egwr99.html
3. Progressive Compression of Visibility Data for View-Dependent Multiresolution Meshes
   Christopher Zach, Konrad Karner
   The 11-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2003, WSCG 2003
   Available Online: http://wscg.zcu.cz/wscg2003/Papers_2003/D47.pdf
4. Efficient Compression of Visibility Sets
   Christian Bouville, Isabelle Marchal, Loic Bouget
   Advances in visual computing : (First international symposium, ISVC 2005) (Lake Tahoe, NV, USA, December 5-7, 2005) (proceedings)
   Lecture Notes in Computer Science (Spinger Verlag), Vol. 3804/2005, pp 243-252